

XOPEN

PORTABILITY GUIDE

SYSTEM V SPECIFICATION
SUPPLEMENTARY DEFINITIONS

XOPEN

PORTABILITY GUIDE
SYSTEM V SPECIFICATION SUPPLEMENTARY DEFINITIONS

3

X/O/P/E/N/

PORTABILITY GUIDE

SYSTEM V SPECIFICATION
SUPPLEMENTARY DEFINITIONS


© 1987, The X/OPEN Group Members

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

X/OPEN PORTABILITY GUIDE

Set of 5 Volumes

ISBN: 0-444-70179-6

Volume 1	XVS Commands and Utilities	ISBN: 0-444-70174-5
Volume 2	XVS System Calls and Libraries	ISBN: 0-444-70175-3
Volume 3	XVS Supplementary Definitions	ISBN: 0-444-70176-1
Volume 4	Programming Languages	ISBN: 0-444-70177-X
Volume 5	Data Management	ISBN: 0-444-70178-8

Published by:

ELSEVIER SCIENCE PUBLISHERS B.V.
P.O Box 1991
1000 BZ Amsterdam
The Netherlands

Sole distributors for the U.S.A. and Canada:

ELSEVIER SCIENCE PUBLISHING COMPANY, INC.
52 Vanderbilt Avenue
New York, N.Y. 10017
U.S.A.

Any comments relating to the material contained in the X/OPEN Portability Guide may be submitted to the X/OPEN Group by letter via the Publisher or directly by Electronic Mail to:

xopen@inset.co.uk

PRINTED IN THE NETHERLANDS



Contents

PREFACE

THE COMMON APPLICATIONS ENVIRONMENT

XVS INTERNATIONALISATION

1. Introduction
2. Overview
3. Native Language Support
4. Message Catalogues
5. Commands
6. Subroutines
7. Header Files

XVS TERMINAL INTERFACE

1. Introduction
2. Curses Overview
3. Data Types and the curses.h Header File
4. Functions
5. Synchronous and Networked Terminals
6. Call Specifications

XVS INTER-PROCESS COMMUNICATION

1. Inter-Process Communication
2. IPC Services

XVS SOURCE CODE TRANSFER

1. Source Code Transfer
2. Floppy Disc
3. Magnetic Tape
4. Utilities
5. Other Techniques



Trademarks

UNIX[™] is a registered trademark of AT&T in the USA and other countries.

C-ISAM[™] is a trademark of Informix Corporation.

LEVEL II COBOL[™] is a trademark of Micro Focus Limited.

XENIX[™] is a trademark of Microsoft Inc.

IBM[™] is a trademark of International Business Machines Corp.

X/OPEN[™] is a licensed trademark of the X/OPEN Group Members.

POSIX[™] is a trademark of the Institute of Electrical and Electronic Engineers Inc.



Preface

X/OPEN represents a major breakthrough in the world of standards for the information technology industry. Ten* of the world's major information system suppliers have come together to encourage applications portability resulting in tangible benefits for computer users, independent software houses and for the suppliers themselves.

The Group's principal aim is to increase the volume of applications available and to maximise the return on investments in software development made by Users and Independent Software Vendors.

This is achieved by ensuring portability of application programs at the source code level. Through this portability, users can mix and match computer systems and applications software from many suppliers, and thus investment in applications software is protected into the future.

In order to provide such portability, the Group defines a **Common Applications Environment** built on the interfaces to the **UNIX** operating system, as defined in the AT&T System V Interface Definition, and covering other aspects required of a comprehensive applications interface.

The X/OPEN Portability Guide contains an evolving portfolio of practical standards for application portability. All of the members of X/OPEN guarantee to support the standards defined, leading to:

- Growing portability
- No dependence on a single source - freedom of choice
- Increased application software selection
- More security in software investments
- International support for the *Common Applications Environment*

X/OPEN is not a standards-setting organisation; it is a joint initiative by members of the business community to integrate evolving standards into a common, beneficial and continuing strategy.

* At the time of publication, the membership of the X/OPEN Group was BULL, DEC, ERICSSON, HEWLETT-PACKARD, ICL, NIXDORF, OLIVETTI, PHILIPS, SIEMENS and UNISYS

Issue 2 of the X/OPEN Portability Guide (published in January 1987) comprises five volumes defining the interfaces currently identified as components of the Common Applications Environment.

Volume 1	System V Specification: Commands and Utilities
Volume 2	System V Specification: System Calls and Libraries
Volume 3	System V Specification: Supplementary Definitions XVS Internationalisation XVS Terminal Interfaces XVS Inter-Process Communication XVS Source Code Transfer
Volume 4	Programming Languages C Language COBOL Language
Volume 5	Data Management Indexed Sequential Access Method (ISAM) Relational Database Language (SQL)

In addition, each volume includes an introduction giving the philosophy of the Common Applications Environment and an overview of its components.

This guide is aimed at both the decision makers and the implementation teams of:

- Independent Software Vendors
- Software Houses
- Users
- Equipment Manufacturers

The Guide is designed to sit permanently on the desk, serving as a common reference point for anyone directly concerned with the practical side of software development, namely systems designers, programmers and consultants.

The various parts of the Portability Guide are closely interrelated. Any reference from one part of the definition to another part uses the title of the other part as a reference (e.g., "XVS TERMINAL INTERFACES").

Acknowledgements

X/OPEN gratefully acknowledges:

- **AT&T** for permission to reproduce portions of its copyrighted System V Interface Definition (SVID) and material from the UNIX System V Release 2.0 documentation.
- The **/usr/group** Standards Committee, whose Standard contributed to the Group's work.
- **Informix Corporation.** of Menlo Park, California (Telex no. 361834) for permission to use material from the specification of their C-ISAM product and for provision of that material in machine readable form.
- **Micro Focus Ltd.** of Newbury, Berkshire for permission to use material from the specification of their LEVEL II COBOL compiler.
- The assistance given by the following companies in the preparation of the Database Language (SQL) definition:

Informix Corporation
Oracle Corporation
Queensland Information Technology
Relational Technology Inc.
Unify Corporation

Referenced Documents

The following documents are referenced in this guide:

- System V Interface Definition (Spring 1985 - Issue 1)
- System V Interface Definition (Spring 1986 - Issue 2)
- UNIX System V Release 2.0 Programmer's Reference Manual (April 1984 - Issue 2)
- UNIX System V - Release 2.0 Programming Guide (April 1984 - Issue 2)
- ANS Draft Proposal for C Language (October 1986 - ANS X3J11/86-151)
- 1984 /usr/group Standard
- IEEE P1003.1 Trial Use Standard (April 1986)
- Informix Corporation C-ISAM Reference Manual (Version 2.10 - January 1985)
- MicroFocus Level II COBOL Language Reference Manual (Version 2.5 and 2.6, Issue 7 - April 1984)
- Standard for COBOL (ANS X3.23-1974)
- Standard for COBOL (ANS X3.23-1985)
- Standard for FORTRAN (ANS X3.9-1978)
- Standard for Database Language (SQL) (ANS X3.135-1986)
- Standard for PASCAL (ISO 7185-1983)

X/O/P/E/N/

PORTABILITY GUIDE

THE COMMON APPLICATIONS
ENVIRONMENT

Contents

Chapter	1	THE COMMON APPLICATIONS ENVIRONMENT
Chapter	2	SYSTEM V
	2.1	INTRODUCTION
	2.2	THE EVOLVING STANDARD
	2.2.1	Origins
	2.2.2	The IEEE "POSIX" Standard
	2.2.3	The AT&T System V Interface Definition
	2.3	THE X/OPEN SYSTEM V SPECIFICATION
	2.3.1	System Calls and Libraries
	2.3.2	Inter-process Communication
	2.3.3	Commands and Utilities
Chapter	3	INTERNATIONALISATION
	3.1	INTRODUCTION
	3.2	The X/OPEN NATIVE LANGUAGE SYSTEM
Chapter	4	C LANGUAGE
	4.1	INTRODUCTION
	4.2	C LANGUAGE PORTABILITY GUIDELINES
	4.3	THE ANS X3J11 DRAFT STANDARD
	4.4	THE C PROGRAM PORTABILITY CHECKER (<i>lint</i>)
Chapter	5	OTHER PROGRAMMING LANGUAGES
	5.1	INTRODUCTION
	5.2	COBOL
	5.3	FORTRAN
	5.4	PASCAL
Chapter	6	DATA MANAGEMENT
	6.1	INTRODUCTION
	6.2	INDEXED SEQUENTIAL ACCESS METHOD (ISAM)
	6.3	RELATIONAL DATABASE LANGUAGE (SQL)

Chapter	7	SOURCE CODE TRANSFER BETWEEN MACHINES
	7.1	INTRODUCTION
	7.2	FLOPPY DISC STANDARD
	7.3	MAGNETIC TAPE
	7.4	UTILITIES
Chapter	8	NETWORKING AND COMMUNICATIONS
	8.1	NETWORKING AND COMMUNICATION
	8.2	OPEN SYSTEMS INTERCONNECTION
	8.3	GENERALISED INTER-PROCESS COMMUNICATION, IPC
	8.4	DISTRIBUTED FILE SYSTEM
	8.5	DISTRIBUTED TRANSACTION PROCESSING

The Common Applications Environment

The formation of the X/OPEN Group represents a major initiative by an international group of suppliers of computer systems to create a free and open market, offering Independent Software Vendors (ISVs) as wide a market as possible for their products and giving users an increased return on investment in application software.

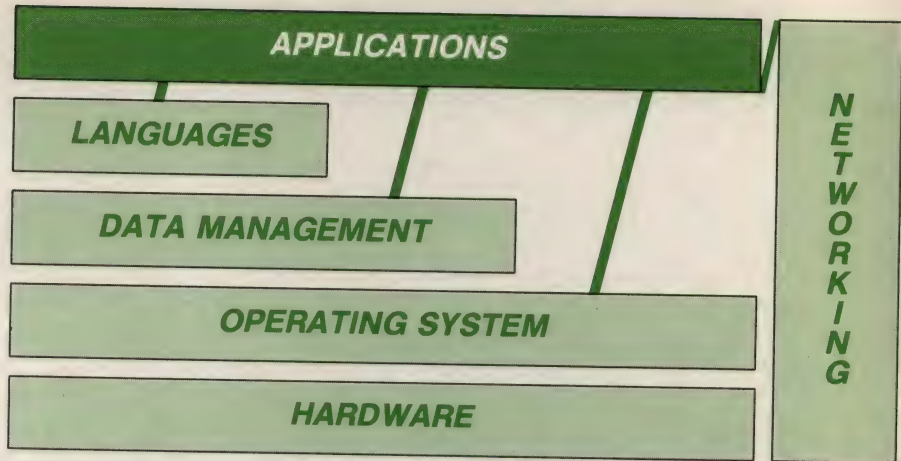
The current dominance of proprietary machine environments is restricting the growth of the computer industry. Users tend to get locked into a particular proprietary system by the investment they have made in the applications. Independent Software Vendors are discouraged from writing applications for a particular environment because of the limited markets caused by this fragmentation. This means that there is very little generally available software for each type of system, thus increasing the size of investment needed by each user. All this in turn limits the sales potential of machines from the computer suppliers.

The objective shared by the members of the X/OPEN Group is to establish a Common Applications Environment to the mutual advantage of users, Independent Software Vendors and computer suppliers. Applications written to operate in this environment will be portable at the source code level to a wide range of machines, thereby releasing the user from dependence on a single supplier, reducing the necessary investment in applications, considerably increasing the market for independent software and opening up the market for systems suppliers.

The existence of these "Open Systems" allows users to mix and match systems from different suppliers, and to move applications between machines to meet changing requirements as business grows, thereby giving protection of investment in applications software into the future.

The great increase in the potential market encourages the Independent Software Vendors to produce a wealth of general applications packages, and the availability of this further reduces the investment needed by the users. The whole situation is thus mutually reinforcing.

The Common Applications Environment



The foundations of the Common Applications Environment are the interfaces of the UNIX System V operating system, as defined in the AT&T "System V Interface Definition", and the C language.

To define a complete environment for portable applications, it is also necessary to satisfy the requirements for data management, integration of applications, data communications, distributed systems, the use of high level languages and the many other aspects involved in providing a comprehensive applications interface. The X/OPEN Group intends, therefore, to publish progressively definitions covering these areas.

The systems of the X/OPEN Group members that support interfaces derived from UNIX operating systems will do this according to the X/OPEN definitions and will support the full Common Applications Environment.

A specific Common Applications Environment feature may not, however, be present if it is not relevant in the market area in which a particular system is sold. For example, a system sold only in a scientific context might not support COBOL. Conversely, a particular system may support features over and above those of the Common Applications Environment, some of which may partially overlap. An example of this could be that an alternative dialect of COBOL is supported in addition to that of the Common Applications Environment.

The X/OPEN Group is primarily concerned with standards selection and adoption. The general policy is to use International Standards, where they exist, and to adopt "de facto" standards in other cases.

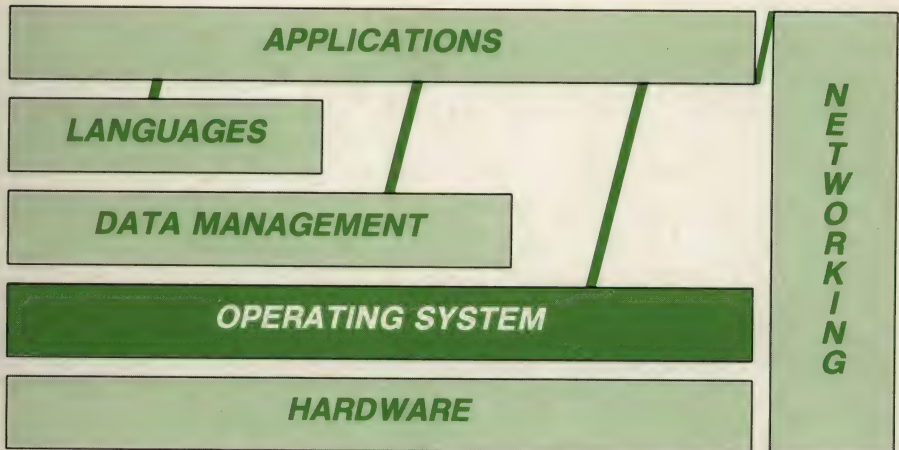
Where International Standards do not exist, it is X/OPEN policy to work closely with standardisation bodies to encourage their emergence.

The Common Applications Environment

It is important that the defined elements of the Common Applications Environment be readily achievable on member systems, and have wide acceptance. For this reason, the definitions, in general, fall within the capabilities of at least one currently available popular product.

In this guide, certain aspects of the Common Applications Environment are defined with reference to the interfaces offered by specific products. This does not mean that member systems will necessarily contain these products, but that the defined interfaces will be supported. Indeed the method of support for an interface on a particular system may change with time.

X/OPEN System V Specification (XVS)



2.1 INTRODUCTION

The X/OPEN System V specification (XVS) defines the applications interfaces provided by the underlying operating system and forms the foundation of the Common Applications Environment.

The XVS is derived from a series of standards activities. The evolving standard is briefly addressed, and the relationship between the X/OPEN System V Specification and the System V Interface Definition and other standards is explained.

2.2 THE EVOLVING STANDARD

2.2.1 Origins

The UNIX operating system was developed by Ritchie and Thompson at Bell Laboratories in the early 1970s. The current AT&T System V version may be traced back directly to that first system.

For many years, it remained basically an academic product. More recently, computer suppliers have adopted the UNIX system as a multi-tasking, multi-user and portable operating environment. They have based their systems on one of several releases, variants or look-alikes. Of these, the most widely used were Version 7, System III, the Berkeley system and XENIX.

Although these systems had much in common, the degree of compatibility at the application interface level was insufficient to permit the development of totally portable applications.

2.2.2 The IEEE "POSIX" Standard

/usr/group, a group of users of UNIX derivatives in the USA, established a committee with the objective of proposing a set of standards for application level interfaces. After publishing its standard, together with a reviewer's guide, the group decided to seek IEEE status for the standard. In late 1984, the /usr/group standards committee closed its activities in its own name and its members were encouraged to become involved in the IEEE group, known as P1003.

The P1003 group published a "trial-use" standard in early 1986, which has the status of a "Draft American National Standard". This "Portable Operating System for Computer Environments" (POSIX) is expected to be revised and submitted for approval in 1987.

The IEEE P1003 group is working to extend the POSIX standard. It is expected that the next area to be standardised will be the subset of commands which offer an interface to applications.

2.2.3 The AT&T System V Interface Definition

The "System V Interface Definition" (SVID), first published by AT&T in the spring of 1985, represented a major standards initiative. AT&T were prominent in the activities of /usr/group and the influence of /usr/group can clearly be seen in the SVID. The stated purpose of the SVID is to define common interfaces for all System V implementations.

The definition groups interfaces into a mandatory *base* plus a series of *extensions*. The *base* interfaces must be present in any implementations of System V. If any interface from an *extension* is supported, it must adhere to the definition.

Issue 1 of the SVID comprised a single volume defining operating system interfaces (known as *system calls* and *library routines*) available to applications as directly called external functions and defined in terms of invocation from C-language programs.

Issue 2 of the SVID was published in early 1986 and comprised two volumes. The first volume contained the same material as issue 1, with some restructuring to improve ease of use and some changes to correct errors. It comprised the *Base System Definition* plus a single extension referred to as the *Kernel Extension*.

The second volume primarily defined commands and utilities, normally invoked through a command interpreter. It comprised further extensions referred to as the *Base Utilities Extension*, *Advanced Utilities Extension*, *Administered Systems Extension*, *Software Development Extension*, and *Terminal Interface Extension*. The latter two include library routines in addition to utilities.

2.3 THE X/OPEN SYSTEM V SPECIFICATION

The X/OPEN System V Specification (XVS) is based upon the AT&T System V Interface Definition, but also taking into account the trial use standard published by IEEE and the capabilities of the existing AT&T System V product.

The XVS is organised into a number of self-contained sections:

"XVS SYSTEM CALLS AND LIBRARIES" defines the Operating System Interfaces and broadly corresponds to Volume 1 of the SVID.

"XVS COMMANDS AND UTILITIES" defines commands and utilities and broadly corresponds to Volume 2 of the SVID. The purpose of the X/OPEN Portability Guide is to facilitate the portability of applications. As such, system administration is outside of its scope and the routines included in the AT&T *Administered System Extension* are not defined.

"XVS TERMINAL INTERFACES" defines a set of portable interfaces to locally connected asynchronous terminals and broadly corresponds to the AT&T Terminal Interface Extension in Volume 2 of the SVID.

"XVS INTER-PROCESS COMMUNICATION" defines interfaces to shared memory, semaphores and message passing, included as an interim mechanism to satisfy the immediate requirements of Inter-Process Communication facilities.

2.3.1 System Calls and Libraries

"XVS SYSTEMS CALLS AND LIBRARIES" contains a full definition of interfaces to system calls and library routines and broadly corresponds to Volume 1 of the SVID.

The X/OPEN Group has extended the SVID in a number of areas:

- Certain changes have been included, which the SVID denotes as future directions.
- The use of symbolic names to replace numeric constants, introduced by AT&T in their SVID, has been extended.
- Clarification of existing wording has been introduced in a limited number of places to "tighten" the specification.
- The opportunity has been taken to correct clerical errors in the SVID.
- Definitions have been included of a number of further UNIX System V Release 2.0 functions which are in widespread use by application developers.

The relationship between the XVS and the SVID is clearly stated. The whole of the SVID *base* definition is included as mandatory with the exception of the maths group, which is not mandatory for systems sold into markets where it is not relevant. *Termio* was not mandatory in issue 1 of the XVS because of some difficulties in implementation. These have now been resolved, and the routine is now mandatory, except in systems which do not support locally connected asynchronous lines.

The XVS incorporates all the interfaces within the SVID *kernel extension set* although a number are defined as optional.

In the XVS, interfaces defined as *optional* will be available on most but not necessarily all X/OPEN systems; use of them could restrict portability. Any *optional* interface supported on an X/OPEN system will conform to the X/OPEN specification.

The XVS defines interfaces in terms of their interface syntax and run-time behaviour, without constraining the method of their implementation. The names "system calls" and "subroutines" are retained purely for compatibility with other documentation.

2.3.2 Inter-process Communication

The kernel extension interfaces relating to shared memory, semaphores and message passing are included in "XVS INTER-PROCESS COMMUNICATION" as a short-term mechanism to satisfy the immediate requirements for Inter-Process Communication facilities. However, they are machine specific and cannot be supported on all hardware architectures. The Group believes that a more generalised approach to the whole subject of Inter-Process Communication is required.

2.3.3 Commands and Utilities

"XVS COMMANDS AND UTILITIES" contains a full definition of interfaces to commands and utilities and broadly corresponds to Volume 2 of the SVID. The interfaces are split functionally into those which are intended to provide an applications interface (referred to as *Standard Utilities*) and those which are only intended to be used by development programmers or during the porting of applications to an X/OPEN system (referred to as *Development Utilities*). The *Standard Utilities* will be present in all X/OPEN systems, as they are needed to provide a run-time interface to applications. The *Development Utilities* may only be present in development systems; their respective descriptions are clearly annotated to indicate this. This same distinction is also present in the SVID. The X/OPEN development utilities correspond exactly to the SVID *Software Development Extension*.

The definition of standards for commands and utilities is an evolving process and X/OPEN intends to participate fully.

The current definition is a valuable first step, but additional work will be needed to evolve towards a complete standard. For example:

- The number of options defined for many of the commands is excessive and includes functionality which is rarely used or is implementation specific.
- There are too many different ways of achieving the same results.
- Many of the current descriptions were written to record the observed behaviour of already existing utilities and the level of precision is inadequate for use as a definitive standard.

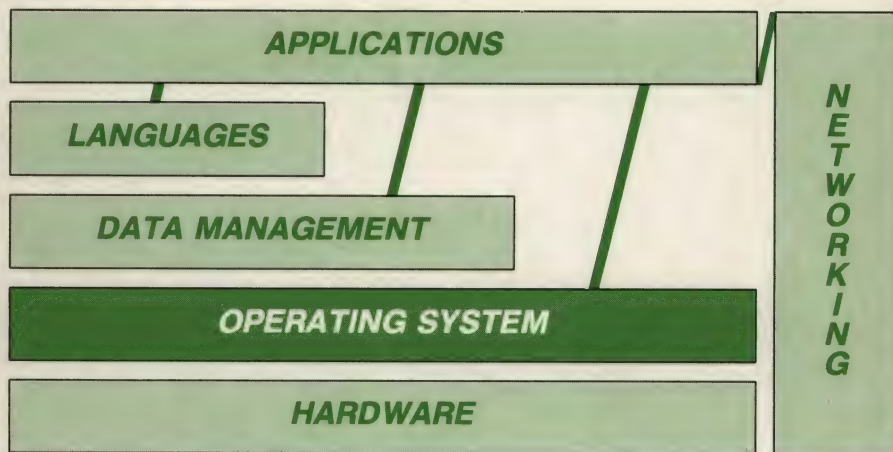
Rectification of this is an enormous task and the current X/OPEN definition is of necessity based on the available documentation, but incorporates extensive annotation to highlight potential portability problems resulting from the points listed above. To achieve maximum portability, application developers should avoid the use of the functionality so annotated.

X/OPEN is working on a substantially improved definition of commands, with the number of options reduced to those in common use and with a higher level of specification. In addition to the current X/OPEN specification, "XVS COMMANDS AND UTILITIES" contains a proposed template for improved specifications together with a number of examples.

This improvement process will be carried out in close consultation with the various user organisations and standards bodies, such as IEEE and ISO, to ensure that the result is a single standard definition of operating system commands and utilities.

Readers of the X/OPEN Portability Guide are invited to participate directly in the consultative process to ensure that the evolving standard matches the requirements of existing and future applications.

Internationalisation



3.1 INTRODUCTION

X/OPEN members market systems in many countries. Our customers and users speak many different languages and conform to different cultural conventions and business practices. It is important therefore that X/OPEN systems are capable of supporting a range of language and cultural environments. In many cases a strong requirement also exists to cope with these variations on the same system. An example is within the administration of the European Economic Community.

To date UNIX operating systems and most systems derived from them have been based on the ASCII 7-bit coded character set and on American English. There are no facilities for dealing with other coded character sets, nor for supporting different languages and cultural conventions.

The requirement for effective mixed language working brings with it the need for coded character sets larger than can be accommodated by 7-bit characters, as does the requirement to support the more complex languages. At the same time there is a trade-off between the ability to handle larger coded character sets, and the amount of storage required to hold the data. For most European requirements an 8-bit system provides the correct balance. For the major Eastern languages (such as Chinese and Japanese) a 16-bit system is necessary, even to support a single language.

To satisfy these requirements enhancements must be made to the system to provide full data transparency to applications, allowing flexibility in the choice of coded character set(s) employed. Additionally, the system must allow program messages (both input and output) to be handled in the native language of each user, as well as providing cultural dependent data items (such as date formats and currency symbols).

3.2 THE X/OPEN NATIVE LANGUAGE SYSTEM

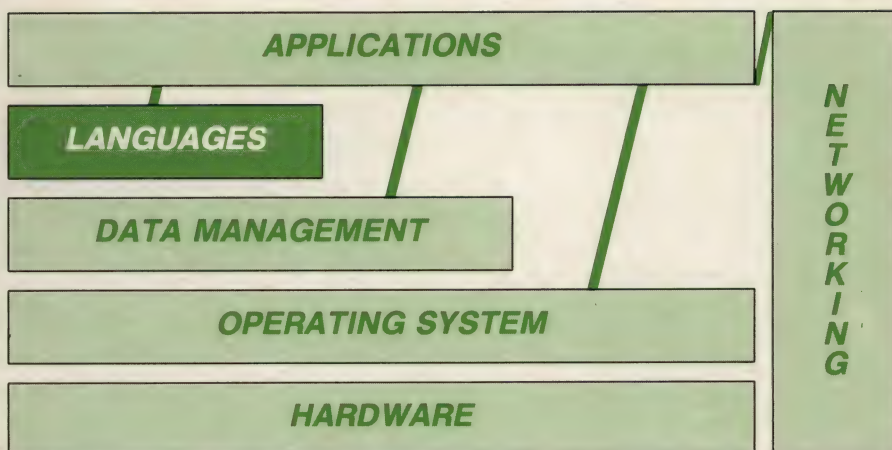
The X/OPEN Native Language System (NLS) is a set of interfaces designed to facilitate the development of applications that can operate in many different language and cultural environments. The interfaces have been derived from those of the Native Language Support system developed by the Hewlett-Packard Company of Palo Alto, California. They have been further enhanced by X/OPEN and have been modified in strategic areas to more closely relate to the Internationalisation proposals of the Draft Proposed American National Standard for the C Programming Language.

The first issue of the specification, defined in "XVS INTERNATIONALISATION", concentrates on facilities for the development of internationalised applications (rather than on internationalising the operating system itself), and on the 8-bit coded character set situations.

The following groups of facilities are defined:

- A message catalogue system which allows program messages to be held apart from the program logic, translated into different native languages, and the appropriate version retrieved by the program at run time.
- An announcement mechanism whereby native language, local custom (territory) and codeset requirements appropriate to each user can be identified to applications at run time.
- Enhanced interface definitions of standard C library functions, which provide language dependent character type classification, upper to lower case and lower to upper case character conversions, date and time messages, floating point to string conversions, and text collation.
- Library functions which allow programs to determine cultural and language specific data dynamically (e.g. the format of date and time strings, weekday and month names, currency symbols, etc.).
- A set of standard commands and library functions which will operate correctly with 8-bit characters.

C Language



4.1 INTRODUCTION

This chapter addresses the C language and guidelines for portability when writing C code.

Currently the American National C Language Standards committee, X3J11, is working towards a standard for the C programming language. The X/OPEN Group is represented on that committee by member companies and intends to adopt the standard, once it has been established as a practical reality.

Meanwhile, the X/OPEN definition included in "C LANGUAGE" is based upon that given in Chapter 2 of the "System V Programming Guide", Release 2.0, published by AT&T.

4.2 C LANGUAGE PORTABILITY GUIDELINES

Whilst the C language provides the basis for applications portability, it is easy to write statements, using valid C constructs, that are machine specific. Care has to be taken when writing programs that are intended to be portable across a range of systems. "C LANGUAGE" includes advice towards ensuring portability.

4.3 THE ANS X3J11 DRAFT STANDARD

The ANS X3J11 standard has been published in a draft form but may still change before it is approved. However, it is already clear that the standard will impose certain restrictions such that programs written to the current C language definition may not work correctly, if the source is later passed through a compiler that supports the ANS standard.

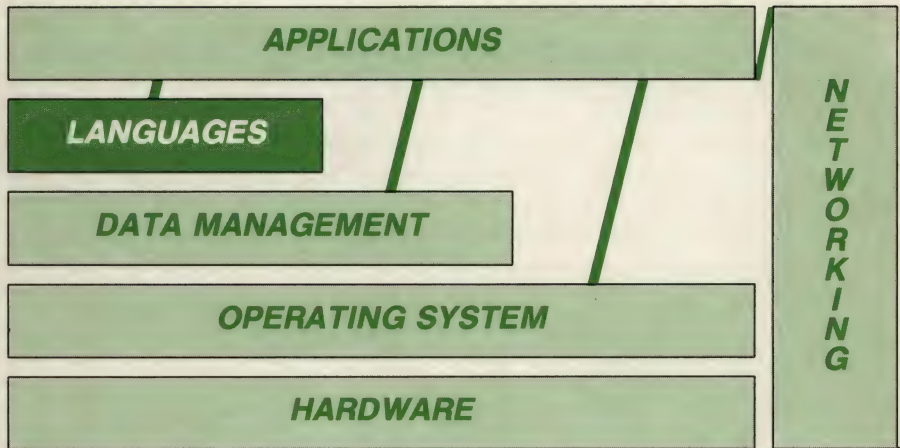
To address this, "C LANGUAGE" includes advice on writing programs to avoid these problems.

4.4 THE C PROGRAM PORTABILITY CHECKER (*lint*)

The *lint* program checks C source programs for violation of most of the portability rules. It also gives a more stringent enforcement of the type rules of C than is provided by most C compilers. A further option detects a number of wasteful or error-prone constructions which nevertheless are syntactically correct.

Use of the *lint* program is recommended: it is described in "C LANGUAGE".

Other Programming Languages



5.1 INTRODUCTION

This chapter addresses programming languages other than C on X/OPEN systems. It covers the inclusion of the principal high level languages in the Common Applications Environment.

To date, The X/OPEN Group has established definitions for COBOL and FORTRAN and PASCAL.

5.2 COBOL

The X/OPEN COBOL definition identifies a common set of language facilities that will be supported by COBOL compilers on all member systems. Applications written to this definition will be portable to any X/OPEN system.

The ISO Working Group and ANS COBOL committee have been working for some years towards a revised standard for COBOL to reflect more accurately the capabilities of modern COBOL compiling systems. The latest international standard, "X3.23 - 1985" was approved during 1985. At the time of publication of Issue 2 of the Portability Guide, there are few compilers in compliance with the revised standard and hence the X/OPEN group feels that major changes in the X/OPEN definition to reflect the new standard would be premature. It is likely, however, that any future edition of the X/OPEN COBOL language definition will relate to "COBOL 1985" and the new standard has been used as a reference when eliminating obsolete elements from the COBOL definition.

The most widely followed standard for COBOL is still that defined in the earlier 1974 Standard, "ANS X3.23-1974", to which most current COBOL compilers substantially conform.

The 1974 standard is incomplete in the area of facilities for interaction with the on-line user. To overcome this deficiency, most COBOL compilers provide extensions to the *ACCEPT* and *DISPLAY* verbs, but they do this in incompatible ways. Since the majority of applications now include interactive operation, it is necessary for a standard form of *ACCEPT* and *DISPLAY* to be defined in the X/OPEN Common Applications Environment.

In order to have an X/OPEN definition that is achievable on member systems within a short timescale, and one that would have immediate widespread acceptance, it has been based on the definition of COBOL embodied in a popular product: Micro Focus LEVEL II COBOL, which itself conforms to the "ANS X3.23-1974".

The Micro Focus LEVEL II COBOL language specification includes a number of other extensions beyond the 1974 standard, in addition to those to *ACCEPT* and *DISPLAY*. None of these are currently included in the X/OPEN definition. The X/OPEN definition also applies restrictions to the ANS-based parts of the LEVEL II definition.

Whilst the X/OPEN COBOL definition is based on the specification of a particular product, the means of implementation across the systems of the X/OPEN members may vary. Any particular system may support extensions beyond the facilities identified, but their use is likely to impede portability.

The X/OPEN COBOL definition is given in detail in "COBOL LANGUAGE" and its relationship to the 1974 standard is clearly shown.

The definition is given in terms of the command syntax derived from the LEVEL II COBOL Reference Manual. The semantics of the *ACCEPT* and *DISPLAY* verbs are defined in "COBOL LANGUAGE". The semantics of all other elements of the language are defined by the "X3.23-1974" standard.

5.3 FORTRAN

The X/OPEN definition for FORTRAN is the formal definition given in the American National Standards document "FORTRAN 77, ANS X3.9 - 1978". This has had wide-scale acceptance throughout the world and there are many certified compilers available.

The majority of FORTRAN compilers, while adhering to the basic FORTRAN 77 standard, also offer extensions beyond that standard. There is little compatibility in these extensions between compilers and they do not form part of the X/OPEN definition. Developers are warned that use of these extensions will affect the portability of FORTRAN programs.

5.4 PASCAL

The current X/OPEN definition for PASCAL is the formal definition given in the International Organisation for Standardisation document "Programming Languages - PASCAL " ISO 7185-1983 (level 1).

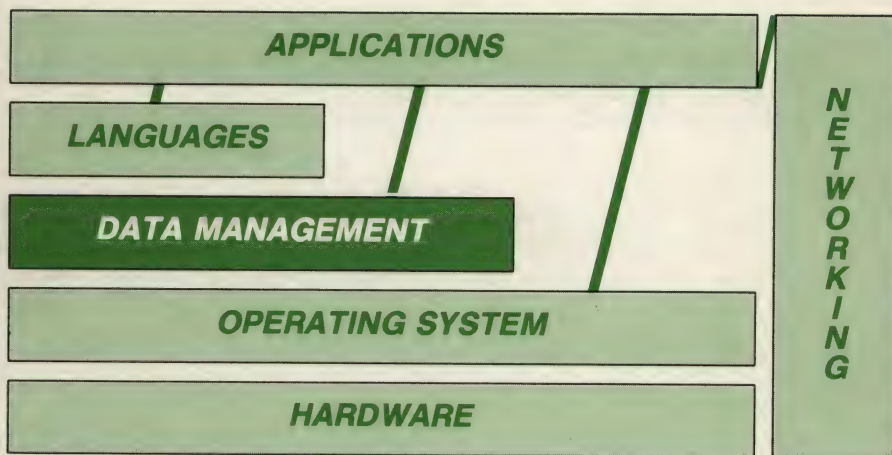
This is well accepted throughout the world and there are significant numbers of certified compilers available.

In order to enhance the portability of programs among PASCAL implementations on X/OPEN systems, the X/OPEN Group has decided to give a uniform definition for certain features designated as implementation-defined in ISO 7185.

- When the required identifiers "input" and "output" occur as program parameters they shall be bound by default to the external system files STDIN and STDOUT respectively.
- There shall be no implementation dependent restrictions on the base-type of a set-type which disallow 0 as the minimal ordinal value and 255 as the maximum ordinal value of that base-type.
- (lazy input) the underlying data transfer action required for a call to the predefined procedure "get" for a textfile (both explicit and where implied by "reset" and "read"), shall be postponed until one of the following events, if any, whichever occurs first :
 - a. access to the buffer-variable of the file
 - b. the buffer-variable of the file is passed as an actual variable parameter to a procedure or function
 - c. a call to the predefined function "eoln" for that file
 - d. a call to the predefined function "eof" for that file

The majority of Pascal compilers also offer extensions beyond the current ISO Standard PASCAL definition. There is little compatibility in these extensions between compilers and developers are warned that use of these extensions may affect the portability of PASCAL programs.

Data Management



6.1 INTRODUCTION

The input/output facilities supported by System V consist only of byte-stream read and write operations on files. No facilities are provided for operating on files as sets of records. This leads to application writers having to make their own arrangements for record handling, resulting in both a multiplication of effort and a proliferation of non-standard methods.

Data Management is a key element in the integration of applications. Applications written in a variety of languages must be able to work on the same basic data in the same form, and data must be passed easily and efficiently between applications.

Addressing these issues, the X/OPEN Group defines interfaces for the creation, management and manipulation of indexed files, generally known as the Indexed Sequential Access Method (ISAM) and for access to relational database management systems, the standard Relational Database Language (SQL).

The availability of these interfaces on X/OPEN systems will not only provide application portability, but will ease and encourage integration.

6.2 INDEXED SEQUENTIAL ACCESS METHOD (ISAM)

The X/OPEN definition for ISAM, which is contained in "INDEXED SEQUENTIAL ACCESS METHOD", is a major subset of the specification of the C-ISAM product, Version 2.10, published by Informix Corporation

The full specification of C-ISAM contains implementation details specific to that product, in addition to the definition of the interface available to applications. Only the applications interface forms part of the X/OPEN definition; implementation details specific to C-ISAM have been omitted. Indeed, there are alternative implementations available on particular member systems.

6.3 RELATIONAL DATABASE LANGUAGE (SQL)

To reflect the growing significance of Relational Data Base systems, the X/OPEN group has defined application interfaces embedded within high level "host" languages to a relational database management system for a free-standing database.

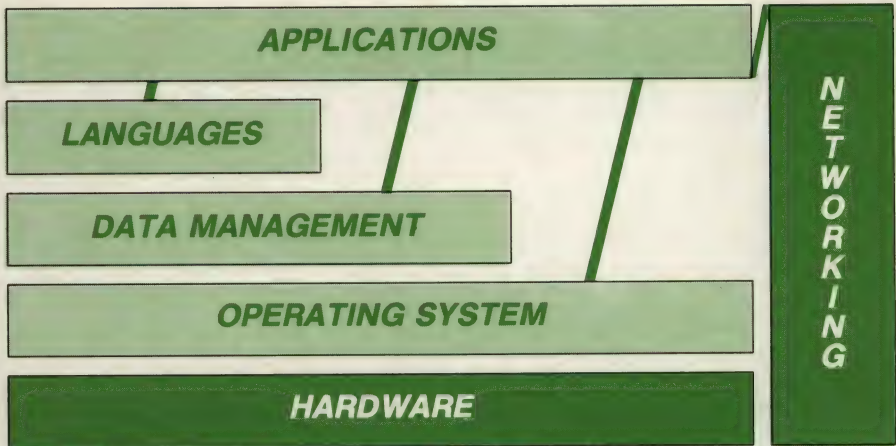
The widely accepted standard for access to relational data base is that defined in the American National Standard document Relational Database Language (SQL) "ANS X3.135-1986".

The X/OPEN definition is based closely on "X3.135-1986" but taking careful account of the capabilities of the leading relational database management systems currently available. The X/OPEN group has worked closely with the vendors of these products throughout.

"X3.135-1986" allows for two levels of compliance, Level 1 and Level 2. Most existing products comply only at Level 1 although it is expected that a significant number of products will have achieved full compliance with Level 2 before the end of 1987. "X3.135-1986" Level 1 SQL is not an adequate definition for application developers, since it leaves too many areas as implementor-defined. In preparing its definition, the X/OPEN group has examined these areas carefully and an agreed X/OPEN approach defined.

The X/OPEN SQL definition is contained in "RELATIONAL DATABASE LANGUAGE (SQL)", and contains a full description of the syntax and semantics of SQL together with a detailed comparison between the X/OPEN definition and the "ANS X3.135-1986" standard.

Source Code Transfer Between Machines



7.1 INTRODUCTION

One of the major problems inhibiting the porting of applications between UNIX system derivatives is that of incompatible media standards and the physical problems of transferring source code in machine readable form.

The X/OPEN Group takes this problem seriously and has agreed common standards for the transfer of source code. Detailed standards are defined in "SOURCE CODE TRANSFER".

Standards are defined for transfer of 5¼" floppy discs and ½" magnetic tape between machines. Because of the different nature of X/OPEN systems, ranging from single user work stations to large mainframes, it is not possible to define formats which are portable across the whole range. Defining standards for both floppy discs and ½" magnetic tape gives the highest practical coverage of systems.

Current differences in the physical recording formats between cartridge tape devices prevents the definition of a standard for this popular medium.

Because of restrictions imposed by existing hardware, some X/OPEN members are not able to support the floppy disc standard.

7.2 FLOPPY DISC STANDARD

As exchange media, the X/OPEN group defines standards for 40 and 80 track floppy discs. It is intended that the prime format should be 80 track, with 40 track retained for compatibility with personal computers. X/OPEN systems equipped only with 80 track disc drives will offer the facility to read 40 track floppy discs by skipping alternate tracks.

7.3 MAGNETIC TAPE

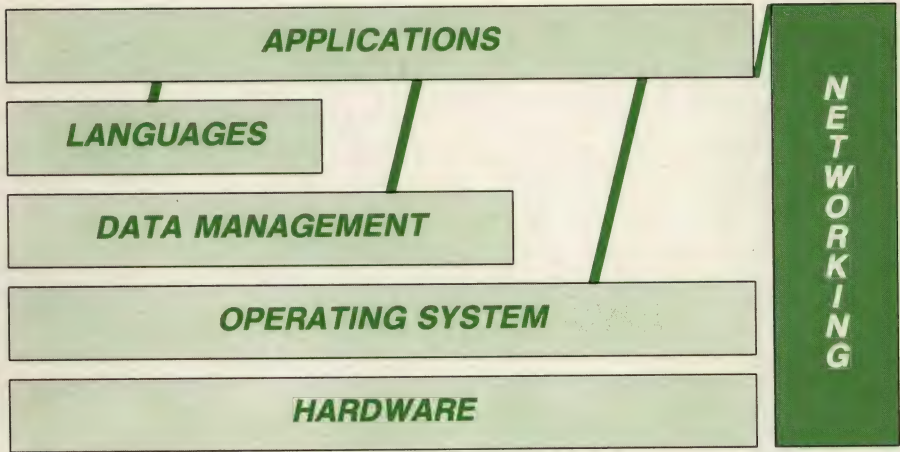
The X/OPEN standard for magnetic tape covers ½" magnetic tape, with a number of different recording formats and densities. The prime format is 9 track Phase Encoded at 1600 bits per inch.

7.4 UTILITIES

"SOURCE CODE TRANSFER" includes the definition of two alternative utilities for the archiving of files to the transfer medium and their subsequent retrieval, *tar* and *cpio*.

In addition, guidelines are given on the use of direct machine to machine connection and the *uucp* utility, as a means of transferring files between X/OPEN systems.

Networking and Communications



8.1 NETWORKING AND COMMUNICATION

The general target for computer data communications is interworking between systems of different types from different suppliers. Future X/OPEN definitions for such open systems interworking can be expected to embrace the ISO OSI standard.

Two services specific to systems supporting the System V Interface have been identified. These are "Generalised Inter Process Communication" (IPC) and "Distributed File System".

Many current commercial applications are supported via interactive transaction processing systems. X/OPEN definitions in this area can be expected to be in line with emerging International Standards.

8.2 OPEN SYSTEMS INTERCONNECTION

For interworking to be possible, systems must have common methods of describing both tasks and data, and must be capable of functioning in a defined manner. To describe interconnection, an architectural model is required. The International Organisation for Standardisation, ISO, has developed the "Reference Model for Open Systems Interconnection" (IS7498). This is often referred to as the "ISO OSI model" or the "ISO 7-layer model". This model sets a framework into which protocol and service standards can be set.

The ISO OSI target is to have a complete set of protocol and service definitions which comply with the 7-layer model and which are internationally agreed and published as ISO standards.

A complete set of ISO OSI standards does not yet exist. Where standards are available, they contain options. For practical systems to be built, it is necessary to have a very clear definition of standards to be adopted and the options to be used.

To provide a clear statement of the standards to be used, a specialist working group has been formed by the 12 major European vendors who propose the technical objectives for "ESPRIT", the "European Strategic Programme for Research into Information Technology". This is called the "Standards Promotion and Application Group, SPAG".

Where ISO standards do not yet exist, interim standards from one of the national or international standards bodies are adopted by SPAG. Such standards are expected to form the basis of future ISO standards. SPAG is not itself a standard making body. Its recommendations will reflect evolving standards.

X/OPEN member companies are committed to the ISO OSI target and the adoption of ISO standards. The group will monitor SPAG recommendations.

X/OPEN intends to define application interfaces for access to OSI services to ensure the portability of applications and library routines.

8.3 GENERALISED INTER-PROCESS COMMUNICATION, IPC

UNIX operating systems provide limited IPC capabilities in the form of "pipes" and "fifos". Kernel extensions within the AT&T System V Interface Definition provide some further IPC mechanisms for the passing of messages between processes in the same memory address space. These extensions were omitted from issue 1 of the X/OPEN definition because it was believed that a much more generalised mechanism for peer to peer communication between processes, either in the same physical machine or in different machines connected via some communications medium is needed.

The X/OPEN group is working on the definition of such a mechanism and but in the short term, it is recognised that there are some applications which need access to such IPC capabilities as currently exist. "XVS INTER-PROCESS COMMUNICATION" gives detailed definitions for

- Message passing between processes.
- Shared memory.
- Semaphores.

It must be recognised that these routines cannot be implemented on all hardware architectures and hence are optional in the X/OPEN definition. However, where the interfaces are supported, the behaviour will be as defined.

8.4 DISTRIBUTED FILE SYSTEM

There is an increasing requirement to be able to access data contained within UNIX File Systems on machines connected together by a local area network from any system on that network. The totality of data accessible in this way can be regarded as a "distributed file system".

The X/OPEN Group regards the way in which local resources are made available to other systems to be a matter of system administration, and does not intend to publish detailed definitions.

The only aspect of such systems which is relevant to the application developer is the behaviour of the system towards applications at run-time.

The characteristics of any distributed file system supported by X/OPEN systems are:

- Access to the distributed file system is via the standard input/output system calls, and is identical for local and remote files.
- No changes are necessary to existing applications. Binary copies of existing applications are able to access a distributed file system, subject to the requirement that the data within a file is in a compatible format.
- Naming of remote items follows the same syntax as for local items and no new naming conventions are required.
- File locking applies across the network.

8.5 DISTRIBUTED TRANSACTION PROCESSING

Many commercial applications require interactive transaction processing facilities and the X/OPEN Group considers the provision of such facilities to be of key importance.

International Standards Organisations have not made the expected progress in this area.

The X/OPEN Group intends to take action to improve this situation.

X/O/P/E/N/

PORTABILITY GUIDE

THE X/OPEN SYSTEM V SPECIFICATION
INTERNATIONALISATION



Contents

Chapter	1	INTRODUCTION
Chapter	2	OVERVIEW
	2.1	GENERAL
	2.2	CHARACTER AND CODE SETS AND TEXT TRANSFER
	2.3	LOCAL CUSTOMS
	2.4	NATIVE LANGUAGE WORKING
	2.5	REGULAR EXPRESSIONS
	2.6	INTERNATIONALISATION AND THE X3J11 DRAFT STANDARD
	2.7	DEFINITIONS
	2.7.1	ASCII
	2.7.2	Character
	2.7.3	Character Set
	2.7.4	Coded Character Set
	2.7.5	Collating Sequence
	2.7.6	Composite Graphic Symbol
	2.7.7	Control Character
	2.7.8	Downshifting
	2.7.9	Graphic Character
	2.7.10	Internationalisation
	2.7.11	ISO 646
	2.7.12	ISO 6937
	2.7.13	ISO 8859/1
	2.7.14	LANG
	2.7.15	Local Customs
	2.7.16	Localisation
	2.7.17	Message Catalogue
	2.7.18	Native Language
	2.7.19	NLSPATH
	2.7.20	Non-spacing Characters
	2.7.21	Radix Character
	2.7.22	Upshifting
Chapter	3	NATIVE LANGUAGE SUPPORT
	3.1	THE NLS DATABASE
	3.1.1	Configuration Data
	3.1.2	Collating Sequence Tables

	3.1.3	Character Classification Tables
	3.1.4	Shift Tables
	3.1.5	Language Information
	3.2	THE ANNOUNCEMENT MECHANISM
	3.3	NLS COMMANDS
	3.4	NLS LIBRARY FUNCTIONS
	3.5	DEFINED CODESETS
Chapter	4	MESSAGE CATALOGUES
	4.1	INTRODUCTION
	4.2	CREATING A MESSAGE CATALOGUE
	4.2.1	Message Text Source Files
	4.2.2	Gencat
	4.3	ACCESSING A MESSAGE CATALOGUE
	4.3.1	Catgets
	4.3.2	Catgetmsg
	4.4	SEARCH AND NAMING CONVENTIONS
Chapter	5	COMMANDS
		<i>gencat(1)</i>
Chapter	6	SUBROUTINES
		<i>catgetmsg(3C)</i>
		<i>catgets(3C)</i>
		<i>catopen(3C)</i>
		<i>conv(3C)</i>
		<i>ctime(3C)</i>
		<i>ctype(3C)</i>
		<i>ecvt(3C)</i>
		<i>nl_init(3C)</i>
		<i>nl_langinfo(3C)</i>
		<i>printf(3C)</i>
		<i>scanf(3C)</i>
		<i>string(3C)</i>
		<i>strtod(3C)</i>
Chapter	7	HEADER FILES
		<i>ctype(5)</i>
		<i>environ(5)</i>
		<i>langinfo(5)</i>
		<i>limits(5)</i>
		<i>nl_types(5)</i>

Introduction

X/OPEN members market systems in many countries. Our customers and users speak many different languages and conform to different cultural conventions and business practises. It is important therefore that X/OPEN systems are capable of supporting a range of language and cultural environments. In many cases, a strong requirement also exists to cope with these variations on the same system. An example is within the administration of the European Economic Community.

To date UNIX operating systems and most systems derived from them have been based on the ASCII 7-bit coded character set and American-English. There are no facilities for dealing with other coded character sets, nor for supporting different languages and conventions.

The requirement for effective mixed language working brings with it the need for coded character sets larger than can be accommodated by 7-bit characters, as does the requirement to support the more complex languages. At the same time there is a trade-off between the ability to handle larger coded character sets, and the inefficiency that can result in the amount of storage required to hold the data. For most European requirements an 8-bit system provides the correct balance. For the major Eastern Languages (such as Chinese and Japanese), a 16-bit system is necessary even to support a single language.

To satisfy these primary requirements, enhancements must be made to the system to provide full data transparency to applications, allowing flexibility in the choice of coded character sets employed. Additionally, the system must allow program messages (both input and output) to be handled in the native language of each user, as well as providing cultural dependent data items (such as date formats and currency symbols).

This part of the Portability Guide defines the X/OPEN Native Language System (NLS) interfaces. They are derived from the interfaces of the Native Language Support system developed by the Hewlett-Packard Company of Palo Alto, California. They have been further enhanced by X/OPEN and have been modified in strategic areas to more closely relate to the Internationalisation proposals of the Draft Proposed American National Standard for the C Programming language.

This first Issue of the definition concentrates on facilities for the development of internationalised applications (rather than on internationalising the operating system itself), and on the 8-bit coded character set situations. It provides a framework within which:

- International programs can be developed to work in many different languages and conform to different cultural conventions.
- The runtime environment of a program can be set up to provide the correct processing/presentation of native language text and cultural data.
- 8-bit coded character sets can be supported. This will be sufficient to meet the requirements of the major Western European Languages.

Unlike most other parts of the X/OPEN Portability Guide, this part describes a new set of facilities which have not had extensive market exposure. Therefore, this first issue should be considered (to some degree) as a trial-use definition, although no changes, other than enhancements, are currently anticipated.

Overview

2.1 GENERAL

The primary function of most computer programs is to manipulate data, some or all of which may involve interaction with the computer user. In commercial situations, it is important that such interactions can take place in the native language of each user. Cultural data should also observe the correct local customs.

Programs intending to support multi-lingual operation must also take into account the fact that languages are represented within a computer system by the characters of one or more coded character sets, which (because of the requirements of different languages) may vary in both size and representation.

Provision for satisfying the above requirements can be made by writing programs which make no hard-coded assumptions about language, local customs or coded character set. Such a program is said to be "internationalised". Data specific to any particular language, cultural convention and codeset are held separate from the program logic, and the process of establishing such data is referred to as "localisation".

The X/OPEN Native Language System (NLS) interface provides definitions of the system facilities which allow the application writer to develop internationalised programs.

Specifically, the following facilities are provided by this issue of the interface definition:

- Message catalogues which allow program messages to be held apart from the program logic, translated into different native languages, and retrieved by the program at runtime.
- An announcement mechanism, whereby native language, local custom (territory) and codeset requirements, as appropriate to each user, can be identified to applications at runtime.
- Internationalised interface definitions of standard C library functions, which provide language dependent character type classification, upper-to-lower case and lower-to-upper case character conversions, date and time messages, floating point to string conversions, and text collation.
- A set of library functions which allow programs to determine cultural and language specific data dynamically (e.g., the format of date and time strings, weekday and month names, currency symbols, radix character symbols, etc.).
- A set of standard commands and library functions which provide 8-bit operation for the processing of data and the names of filestore objects (e.g., files, directories, etc.). For convenience, this is referred to as "8-bit transparency" elsewhere in the interface definition.

In the remainder of the document, the term "language" is often used on its own to identify (collectively) language, territory and codeset. This is done to avoid unnecessarily long phrases in the documentation when any distinction between these items is unimportant.

The rest of this chapter discusses various aspects of the above provisions in greater detail. The remaining chapters constitute the detailed X/OPEN NLS interface definition.

2.2 CHARACTER AND CODE SETS AND TEXT TRANSFER

To date, UNIX operating systems and most systems derived from them have been based on the ASCII 7-bit coded character set, with no provision being made for handling alternative sets. Moreover many parts of the system have relied on the 7-bit scope of the character and have used the eighth bit for other purposes, thus preventing the use of larger codesets, even those that include 7-bit ASCII within them.

Most native languages require characters additional to those catered for by the 7-bit ASCII set, and several require an alternative (e.g., Arabic), and sometime very large (e.g., Japanese), set of characters. Also it is often a requirement to be able to support several languages concurrently on the same computer system, which means that either the codeset in use has to cover the necessary range of languages or that the same system can operate simultaneously with more than one codeset.

There is a comprehensive set of International Standard Codesets covering the major languages, that include the 7-bit ASCII codeset within each member. Several of these cater for the mixing of major languages within a single codeset, for example ISO 8859/1 which covers the major Western European languages in a single 8-bit set. It is therefore believed to be acceptable, and indeed desirable, to define a system that retains the dependence on the 7-bit ASCII encodings within otherwise unrestricted 8-bit (or 16-bit or greater) codesets.

Although it is a primary objective that the definition allows the use of different coded character sets, decisions have to be made as to the actual codeset(s) to be used on a specific system. Consideration also has to be given to the transmission of information between systems. Information must be transferred in a form understandable by the communicating systems, and it is undesirable to have to carry out unnecessary conversions between internal and external (transmission) codes. It is therefore intended to identify specific coded character sets for text interchange between X/OPEN systems in major geographical groupings (e.g., Western Europe), and to "recommend" their adoption also as internal codes.

It is allowable for systems having over-riding constraints (for example the need to be compatible with an existing field population of systems) not to adopt the internal code recommendation, but such systems must provide conversion facilities when communicating with other X/OPEN systems.

Conversion facilities may be necessary for text interchange between geographic groupings (different transmission codes) but in general interchange is only possible by restricting to the set of characters common to both codesets. (The real requirement is of course language translation.) In some situations such conversion may be needed for interchange between parts of the same system (i.e., two internal and external codes in use).

This first issue of the X/OPEN NLS specification defines the major transmission codeset for Western European use as the ISO standard IS8859/1, and also recommends its use as the corresponding internal codeset. Other codesets will be identified in later issues.

(Note: it is anticipated that other significant transmission codes may also have to be supported in specific circumstances. A particular case is ISO 6937 which is the normal transmission code for X.400 and ISO MOTIS mail systems. In general it is not suitable for use as an internal code for data processing due to the extensive use of non-spacing characters.)

2.3 LOCAL CUSTOMS

Conventions for representing cultural data can vary from one country to another, and from region to region within a single country. For example:

- The representation of numbers varies with respect to the symbol indicating the radix character and the digit grouping symbol, e.g., England and France both represent numbers using decimals and commas, but the symbols are interchanged (2,345.77 vs. 2.345,77).
- Currency symbols vary both in terms of the character or characters used and with regards to the position of those characters within a numeric value - i.e., they can precede, follow or appear within the number.
- Ordering of the year, month and day, as well as the separating delimiters, is not universally defined - e.g., October 7, 1986 would be represented as 10/7/1986 in the U.S., 7.10.1986 in Germany, and 1986/10/7 in Japan.

To meet the requirements of local customs, the interface provides a set of library functions which allow cultural data appropriate to the user to be determined at runtime.

2.4 NATIVE LANGUAGE WORKING

Another of the major requirements for internationalisation is to allow programs to communicate with users in their own language. To achieve this, the interface definition must provide the means of defining program messages separately from the computer language source of the program, and runtime facilities for binding a program to the appropriate (native language) instance of its message text.

To this must be added the general requirements that message text should be portable from one system to another, and that program performance must not be seriously impaired by the messaging system.

The following provisions are made in this issue of the interface definition to satisfy the above requirements:

- A formal syntax definition of message text source files, which can be used to describe individual program messages and (where applicable) multiple sets of program messages for different uses. One such source file would normally be defined for each supported language (i.e., where the message text has been translated into the appropriate native language). Message text source files are also defined to be portable between machines.
- The means of generating message catalogues from message text source files. Message catalogues underpin the implementation of various library functions (see below) and are defined to provide a flexible and efficient structure for the internal processing of messages and message sets.
- A set of library functions which allow application programs to locate the required instance of a message catalogue and to extract specific program messages therefrom at runtime.

2.5 REGULAR EXPRESSIONS

It is highly desirable that regular expressions of the type described in *regex(3X)* are defined to work correctly for international use. However, the situation is complicated and consideration of regular expressions is not yet complete. Revised definitions are not therefore included in this first issue of the interface definition.

2.6 INTERNATIONALISATION AND THE X3J11 DRAFT STANDARD

The American National Standards Committee X3J11 Draft Proposed Standard for the C Programming Language includes a number of library routines that have been defined to operate internationally; that is, they modify their operation in a manner appropriate to the native language and cultural environment of the user. There are some areas of overlap with the X/OPEN Internationalisation facilities.

The X/OPEN definition attempts to respect and maintain compatibility with the X3J11 in two important areas:

- The overall style of the syntax
- Where there is a direct correspondence between the library routines in both systems.

In these areas, it is believed that the X3J11 proposal is unlikely to change as a result of the public review process.

In other areas, the X3J11 proposal has introduced new or alternative routines. It is considered that these are more liable to change, and it would be premature for the X/OPEN definition to adopt them.

Applications writers should, however, take note of the areas of difference between the X/OPEN and X3J11 definitions in order to minimise any later portability problems. A summary of the situation is as follows:

1. General architecture and syntactic style is the same, so program structure should not be impacted.
2. The routines listed below are included in both systems and, within the degree of detail available in the X3J11 proposal at the time of writing, they are believed to be substantially compatible.

isalnum	toupper	atof
isalpha	tolower	strtod
isctrl		
isdigit	printf	
isgraph	fprintf	
isupper	sprintf	
isprint	scanf	
isspace	fscanf	
ispunct		

3. The X/OPEN enhanced versions of formatted input and output, which allow for variable ordering of parameters, do not have equivalents in the X3J11 proposal.

<code>nl_printf</code>	<code>nl_fprintf</code>	<code>nl_sprintf</code>
<code>nl_scanf</code>	<code>nl_fscanf</code>	<code>nl_sscanf</code>

4. The methods of character ordering and collating are different. X/OPEN has enhanced versions of *strcmp* (*nl_strcmp*) and *strncmp* (*nl_strncmp*). X3J11 introduces the routine *strcoll*, which is used to preprocess data before calling the standard routine.
5. There are different routines for handling time and date. X/OPEN uses an enhanced form of *ctime* (*nl_cxtime*) and *asctime* (*nl_ascxtime*). X3J11 introduces the routine *strftime*.
6. The announcement mechanisms for introducing the user requirements are different.

2.7 DEFINITIONS

A number of general terms are used in the X/OPEN NLS interface definition. These are defined as follows:

2.7.1 ASCII

American Standard Code for Information Interchange. ASCII is the traditional System V coded character set and defines 128 characters, including both control characters and graphic characters, represented by 7-bit binary values (see also ISO 646).

2.7.2 Character

A member of a set of elements used for the organisation, control or representation of text.

2.7.3 Character Set

A set of alphabetic or other characters used to construct the words and other elementary units of a native language or a computer language.

2.7.4 Coded Character Set

A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and their bit representations.

2.7.5 Collating Sequence

The ordering sequence applied to characters or a group of characters when they are sorted.

2.7.6 Composite Graphic Symbol

A graphic symbol consisting of a combination of two or more other graphic symbols in a single character position, such as a diacritical mark and a basic letter.

2.7.7 Control Character

A character, other than a graphic character, that affects the recording, processing, transmission or interpretation of text.

2.7.8 Downshifting

The conversion of an upper-case character to its lower-case representation.

2.7.9 Graphic Character

A character, other than a control character, that has a visual representation when hand-written, printed or displayed.

2.7.10 Internationalisation

The provision within a computer program of the capability of making itself adaptable to the requirements of different native languages, local customs and coded character sets.

2.7.11 ISO 646

ISO 7-bit coded character set for information interchange. The reference version of ISO 646 contains 95 graphic characters, which are identical to the graphic characters defined in the ASCII coded character set.

2.7.12 ISO 6937

ISO 7-bit or 8-bit coded character set for text communication using public communication networks, private communication networks, or interchange media such as magnetic tapes and discs.

2.7.13 ISO 8859/1

ISO 8-bit single-byte coded character set Part 1, Latin Alphabet No. 1. The ISO 8859/1 character set comprises 191 graphic characters covering the requirements of most of Western Europe.

2.7.14 LANG

An environment variable (*LANG*uage), which is used to announce the user's requirements for native language, local customs and coded character set to the computer system.

2.7.15 Local Customs

Refers to the conventions of a geographical area or territory for such things as date, time and currency formats.

2.7.16 Localisation

The process of establishing the runtime environment of an internationalised computer program to meet the requirements of particular native languages, local customs and character sets.

2.7.17 Message Catalogue

A file or storage area containing program messages, command prompts, and responses to prompts for a particular native language, territory and codeset.

2.7.18 Native Language

A computer user's spoken or written language, such as Dutch, English, French, German, Italian, Swedish, American-English, etc..

2.7.19 NLSPATH

An environment variable used to indicate the search path for message catalogues.

2.7.20 Non-spacing Characters

A character, such as a character representing a diacritical mark in the ISO 6937 coded character set, which is used in combination with other characters to form composite graphic symbols.

2.7.21 Radix Character

The character that separates the integer part of a number from the fractional part.

2.7.22 Upshifting

The conversion of a lower-case character to its upper-case representation.

Native Language Support

3.1 THE NLS DATABASE

The X/OPEN NLS interface defines the functional capabilities of a generic database, which is used to hold various language dependent entities such as:

- Configuration data
- Collating sequence tables
- Character classification tables
- Shift tables
- Language information
- Message catalogues

The first five items in the above list are described in more detail in the following sections. Message catalogues are described in chapter 4.

3.1.1 Configuration Data

Configuration data identifies the languages supported on a system in terms of valid language/territory/codeset settings. Each supported language will have its own set of collating sequence, character classification and shift tables, language information and message catalogues.

3.1.2 Collating Sequence Tables

These tables define the collating sequence for each supported language. The binary values of characters in the associated coded character-set are used as indices into the table, individual entries of which indicate the relative position of that character in the language collating sequence. The interface definition provides for support of the following capabilities:

- 1-to-1 character mappings
- 1-to-2 character mappings, where certain characters require to be treated as if they were two characters (e.g., a sharp "s" in German, which becomes "ss").
- 2-to-1 character mappings, where certain character sequences require to be treated as if they represented a single character in the collating sequence (e.g., "ch" and "ll" in Spanish, which are collated after "c" and "l" respectively).
- Don't-care characters, where certain characters require to be ignored by the collating sequence. For example, if "-" were defined as a don't-care character, then the strings "re-locate" and "relocate" would be equal.

These capabilities extend to providing support for collating algorithms which cater for case and accent priority, e.g., where two characters are first compared for equality ignoring accents, and if equal are then ordered by accent sequence. Collating algorithms of this type provide for the "dictionary" ordering of data (see *string(3C)*).

3.1.3 Character Classification Tables

These contain the lookup tables for character classification. Each character code from the defined coded character set is used as an index into the relevant language lookup table, each entry of which contains a series of flags identifying the truth or falsehood of a particular language assertion, such as:

- undefined character
- upper-case alphabetic
- lower-case alphabetic
- decimal digit
- punctuation
- control
- blank
- hexadecimal digit

3.1.4 Shift Tables

Shift tables contain the corresponding upper-case and lower-case bit combinations for each character. Thus, the upshifted or downshifted value of a character can be determined by accessing the relevant character entry in the shift table.

3.1.5 Language Information

Language information (or *langinfo*), contains the message text for each of the constants defined in *langinfo.h*. The function *nl_langinfo(3C)* provides a procedural interface to this data, allowing applications to discover cultural and language specific information at runtime, such as:

- Date and time formats
- Days of the week
- Months of the year
- Abbreviated names of days of the week
- Abbreviated names of months of the year
- Radix character
- Separator for thousands
- Affirmative and negative responses for yes/no questions
- Currency Symbol and its position within a currency value.

3.2 THE ANNOUNCEMENT MECHANISM

An environment variable *LANG* is defined to identify the user's localisation requirements with regards to language, territory and codeset, where a unique value of *LANG* is defined for each supported language/territory/codeset combination. Each *LANG* setting has associated with it instances of collating sequence, character conversion, character classification and *langinfo* tables, and message catalogues.

LANG contains the required language, territory and codeset names in English as follows:

```
language[_territory[.codeset]]
```

where the length of the entire string should not exceed `{NAME_MAX}` characters, and the set of characters, excluding separators, is restricted to the ASCII set of alphanumeric characters.

On its own, language selects the required native language and implies the default *langinfo* data and the default codeset tables for that language. These defaults can be established separately on each system, such that (for example) the default *langinfo* data for the language designated "French" might contain French cultural data in France and Swiss cultural data in Switzerland.

If other than the default settings are required, then either `"_territory"` or `"_territory.codeset"` can also be specified. For example:

```
LANG=french
```

or

```
LANG=french_swiss
```

or

```
LANG=french_swiss.6937
```

LANG can be set up on a system-wide basis in `/etc/profile`, or for each user individually in `$HOME/.profile` (see "XVS COMMANDS AND UTILITIES", *sh*(1)).

At runtime, the end user's requirements for language, territory and codeset, as defined by the current setting of *LANG*, can be bound to the operation of a program by calling *nl_init*. This function initialises NLS operation by loading the relevant lookup tables into mainstore. Further calls to *nl_init* can be used by an application program to switch between one supported language and another.

(Note: It is not essential that announcement is always made via the *LANG* environment variable. Application programs can derive language, territory and codeset settings to be passed as the parameter to call of *nl_init* from any appropriate source.)

3.3 NLS COMMANDS

The following standard commands are identified as the minimum set of commands which will provide 8-bit transparency on all conforming systems:

Interface	Entry	Interface	Entry
cat	<i>cat</i> (1)	mkdir	<i>mkdir</i> (1)
cd	<i>cd</i> (1)	mv	<i>cp</i> (1)
cp	<i>cp</i> (1)	pr	<i>pr</i> (1)
cpio	<i>cpio</i> (1)	red	<i>ed</i> (1)
echo	<i>echo</i> (1)	rm	<i>rm</i> (1)
ed	<i>ed</i> (1)	rmdir	<i>rm</i> (1)
expr	<i>expr</i> (1)	sed	<i>sed</i> (1)
find	<i>find</i> (1)	sh	<i>sh</i> (1)
grep	<i>grep</i> (1)	sort	<i>sort</i> (1)
ln	<i>cp</i> (1)	tr	<i>tr</i> (1)
ls	<i>ls</i> (1)	wc	<i>wc</i> (1)

Note that the provision of message catalogues for these commands, and their correct operation with respect to local customs and regular expressions, is not covered by this issue of the interface definition.

In addition, the NLS interface defines a message catalogue system command *gencat*, which is used for generating native language message catalogues from message text source files provided by the application writer. Message catalogues, message text source files and the *gencat* command are defined in chapter 4.

3.4 NLS LIBRARY FUNCTIONS

The following library functions are defined for use by internationalised application programs

Interface	Entry
atof	<i>strtod</i> (3C)
catclose	<i>catopen</i> (3C)
catgetmsg	<i>catgetmsg</i> (3C)
catgets	<i>catgets</i> (3C)
catopen	<i>catopen</i> (3C)
fprintf	<i>printf</i> (3S)
fscanf	<i>scanf</i> (3S)
gcvt	<i>ecvt</i> (3C)
isalnum	<i>ctype</i> (3C)
isalpha	<i>ctype</i> (3C)
iscntrl	<i>ctype</i> (3C)
isdigit	<i>ctype</i> (3C)
isgraph	<i>ctype</i> (3C)
islower	<i>ctype</i> (3C)
isprint	<i>ctype</i> (3C)
ispunct	<i>ctype</i> (3C)
isspace	<i>ctype</i> (3C)
isupper	<i>ctype</i> (3C)
isxdigit	<i>ctype</i> (3C)
nl_ascxtime	<i>ctime</i> (3C)
nl_cxtime	<i>ctime</i> (3C)
nl_fprintf	<i>printf</i> (3S)
nl_fscanf	<i>scanf</i> (3S)
nl_init	<i>nl_init</i> (3C)
nl_langinfo	<i>nl_langinfo</i> (3C)
nl_printf	<i>printf</i> (3S)
nl_scanf	<i>scanf</i> (3S)
nl_sprintf	<i>printf</i> (3S)
nl_sscanf	<i>scanf</i> (3S)
nl_strcmp	<i>string</i> (3C)
nl_strncmp	<i>string</i> (3C)
printf	<i>printf</i> (3S)
scanf	<i>scanf</i> (3S)
sprintf	<i>printf</i> (3S)
sscanf	<i>scanf</i> (3S)
strtod	<i>strtod</i> (3C)
tolower	<i>conv</i> (3C)
toupper	<i>conv</i> (3C)

3.5 DEFINED CODESETS

The following codesets are currently specified:

Major Western European transmission set: IS8859/1.

ISO 8-bit Single Byte Coded Graphic Set Part 1: Latin Alphabet No 1.
(Comprises 191 graphic characters.)

Recommended internal codeset for major Western European use: IS8859/1.

Other codesets will be specified in future issues.

Message Catalogues

4.1 INTRODUCTION

The message catalogue system allows program messages to be stored separate from the logic of a program, translated into several languages, and retrieved at runtime according to the language requirements of each user.

To facilitate this process, the interface definition provides the command:

- *Gencat*, which produces a message catalogue from one or more message text source files.

and the following library functions for locating and accessing message catalogues at runtime:

- *Catopen*, which locates a named message catalogue in the filestore and prepares it ready for use by *catgets*, *catgetmsg* and *catclose*.
- *Catgetmsg* and *catgets* which retrieve messages from a message catalogue opened by a call to *catopen*.
- *Catclose*, which closes a message catalogue.

Optionally, message catalogues can be divided into one or more sets of program messages, each set containing one or more numbered messages. Both *catgetmsg* and *catgets* allow programs to access numbered messages within numbered message sets.

4.2 CREATING A MESSAGE CATALOGUE

The steps that a programmer should take in order to produce internationalised application programs are:

- Separate the program logic from program messages by using calls to *catopen*, *catgets*, *catgetmsg* and *catclose* to retrieve message text at runtime, instead of hard-coding messages into the source program.
- Create a message text source file for localisation using any standard text editor.
- Generate a message catalogue from the message text source file using *gencat*.

Further localisations could then be provided by translating the strings contained in the message text source file into different native languages, and by using *gencat* to create the various native language message catalogues.

4.2.1 Message Text Source Files

The format of a message text source file is defined as follows. Note that the fields of a message text source line are separated by a single ASCII space or tab character. Any other ASCII spaces or tabs are considered as being part of the subsequent field.

\$set*n comment*

This line specifies the set number of the following messages until the next *\$set*, *\$delset*, or end-of-file appears. The *n* denotes the set number, which is defined as an integer value in the range (1-{NL_SETMAX}). Set numbers must be presented in ascending order within a single source file but need not be contiguous. Any string following the set number is treated as a comment. There must be at least one *\$set* directive in a message text source file.

(The use of message sets is not defined in this issue of the interface definition. As a future direction, message sets are likely to be used to help improve performance of the message catalogue system by grouping individual messages into message sets according to their frequency of use. This would allow message sets designated as "frequently used" to be located in mainstore by (for example) the *catopen* function.)

\$delset*n comment*

This line deletes the entire message set from an existing message catalogue. The *n* denotes the set number (1-{NL_SETMAX}). Any string following the set number is treated as a comment.

\$*comment*

A line beginning with a dollar symbol (\$) followed by an ASCII space or tab character is treated as a comment.

m*message-text*

The *m* denotes the message number, which is defined as an integer value in the range (1-{NL_MSGMAX}). *Message-text* is stored in the message catalogue with the set number specified by the last *\$set* directive, and with message number *m*. If the *message-text* is empty, then conceptually a null

string is stored in the message catalogue, which (in effect) allows the text of existing messages to be deleted. Note that *catgets* and *catgetmsg* do not distinguish between a null message and an undefined message, i.e., in both cases, these functions will return a pointer to the null string. Message numbers must be in ascending order within a single set but need not be contiguous. The length of *message-text* must be in the range (0 {NL_TEXTMAX}).

(As a future direction, it is intended to allow *m* to be supplied as a symbolic name which can also be specified as an argument to *catgetmsg* and *catgets*.)

\$quote *c*

This line specifies an optional quote character *c*, which can be used to surround *message-text* so that trailing spaces are visible in a message source line. By default, or if an empty *\$quote* directive is supplied, no quoting of *message-text* will be recognised.

Empty lines in a message text source file are ignored.

Text strings can contain the special characters and escape sequences defined in the following table:

Description	Symbol	Sequence
newline	NL(LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
bit pattern	ddd	\ddd

The escape sequence *\ddd* consists of backslash followed by 1, 2 or 3 octal digits, which are used to specify the value of the desired character. If the character following a backslash is not one of those specified, the backslash is ignored. Backslash, **, is also used to continue a string on the following line. Thus, the following two lines describe a single message string:

```
1 This line continues \
  to the next line
```

which is equivalent to:

```
1 This line continues to the next line
```


4.2.2 Gencat

Gencat takes a message text source file and produces either a new message catalogue, or merges the new message text into an existing message catalogue. For example:

```
gencat catfile msgfile
```

where *catfile* is the name of the target message catalogue and *msgfile* is the name of a message text source file. If *catfile* exists, then the messages and sets defined in *msgfile* will be added to *catfile*. If set and message numbers collide, the new message text given in *msgfile* will replace the existing message-text contained in *catfile*. If *catfile* does not exist, it will be created.

Note that no naming conventions are defined for either message text source files or message catalogues.

4.3 ACCESSING A MESSAGE CATALOGUE

Message catalogues are opened ready for use by calling the library function *catopen*, which locates the identified message catalogue according to the search and naming rules defined in the environment variable *NLSPATH* (see the section **Search and Naming Conventions**). For example:

```
nl_catd = catopen (argv[0], 0);
```

If successful, *catopen* returns a catalogue-descriptor *nl_catd* (see *nl_types(5)*), which is used on subsequent calls to *catgets* and *catgetmsg* to identify the prepared message catalogue. Message catalogues are closed by calling the library function *catclose*.

4.3.1 Catgets

Catgets retrieves a numbered message from a numbered message set in the message catalogue identified by *catd*:

```
char *catgets (catd, set_num, msg_num, s)
```

where *set_num* is the number of the message set containing the message *msg_num*, and *s* is a pointer to the default message string. If the message is retrieved successfully, then a pointer to the message text is returned to the caller. If the call is unsuccessful because the message catalogue identified by *catd* is not available, then a pointer to *s* is returned. Otherwise, if *msg_num* is not contained in the message catalogue identified by *catd*, *catgets* returns a pointer to the null string.

All buffer handling and storage space allocation (for holding the text of a program message) are performed by *catgets* internally.

For example, the following C source program uses *catopen* and *catgets* to retrieve messages from the message catalogue identified as *prog*:

```
#include <stdio.h>
#include <nl_types.h>
#define NL_SETN 1

main ()
{
    nl_catd catd = catopen ("prog", 0);
    printf ("%s\n", catgets (catd, NL_SETN, 1, "hello world"));
    catclose (catd);
}
```


Default message strings provide for one set of localisations being kept with the program as an aid to readability. Alternatively, they can be used to allow application programs to continue working predictably when specific localisations of the message text are not currently available. For example, if the above program were invoked from the command level as follows;

```
$ LANG=spanish prog
```

and assuming that *spanish* message text for *prog* was not currently defined on the system, then the above invocation of *prog* would cause the default message string to be displayed, i.e.,

```
hello world
$
```

4.3.2 Catgetmsg

Catgetmsg retrieves a numbered message from a numbered message set in the message catalogue identified by *catd*:

```
char *catgetmsg (catd, set_num, msg_num, buf, buflen)
```

where *catd* is a catalogue descriptor returned by a call to *catopen*, which refers to the message catalogue containing the identified message set (*set_num*) and the required message (*msg_num*). *Buflen* is an integer value containing the size in bytes of *buf*. *Catgetmsg* will read at most *buflen-1* bytes into *buf*. The message string is then terminated with a null byte.

Note that the length of messages may vary considerably between languages, resulting in the silent truncation of long messages if *buflen* is less than the length of the identified message string

The C example given in the previous section could thus be rewritten as follows to use *catgetmsg*:

```
#include <stdio.h>
#include <nl_types.h>
#define NL_SETN 1

char buf[BUFSIZ];

main ()
{
    nl_catd catd = catopen ("prog", 0);
    printf ("%s\n", catgetmsg (catd, NL_SETN, 1, buf, BUFSIZ));
    catclose (catd);
}
```

In this case, if the message catalogue is not available, or the identified message cannot be retrieved, *catgetmsg* will return a pointer to an empty string. Thus using the previous example, and assuming that the same conditions apply, invoking the above variant of *prog* would have the following results:

```
$ LANG=spanish prog
```

```
$
```


4.4 SEARCH AND NAMING CONVENTIONS

The names of message catalogues, and their location in the filestore, can vary from one system to another. Filename suffixes are not universally defined, and individual applications can choose to name or locate message catalogues according to their own special needs. For example:

- Message catalogues for the standard commands might be located in a central library of catalogues containing, for each supported language, a directory and the corresponding message localisations for each command (e.g., in `/ <path-prefix> / $LANG / *.cat`).
- Applications, on the other hand, might find this structure inconvenient, preferring instead to locate their message catalogues in a single program directory, with one message catalogue per supported language (e.g., in `/ <path-prefix> / prog / $LANG`).

These requirements are satisfied by defining an NLS environment variable *NLSPATH*, which gives both the location of message catalogues, in the form of a search path, and the naming conventions associated with message catalogue files. For example:

```
NLSPATH= /nlslib/%L/%N.cat:/nlslib/%N/%L
```

The metacharacter `%` introduces a substitution field, where `%L` substitutes the current setting of *LANG*, and `%N` substitutes the value of the *name* parameter passed to *catopen*. Thus, in the above example, *catopen* will search in `/nlslib/$LANG/name.cat`, then in `/nlslib/name/$LANG`, for the required message catalogue.

NLSPATH will normally be set up on a system wide basis (e.g., in `/etc/profile`) and thus makes the location and naming conventions associated with message catalogues transparent to both programs and users. For a complete description of *NLSPATH* see *environ*(5).

Commands

This section contains a description of the NLS command *gencat*, which is used for creating and maintaining message catalogues. The format of the message text source files processed by *gencat* is defined in chapter 4.

In addition, the following are identified as the minimum set of commands which are guaranteed to provide full 8-bit transparency on all conforming X/OPEN systems. Note that the definitions of these commands, in terms of their syntax and parameters, is not changed by the operation of NLS.

Interface	Entry	Interface	Entry
cat	<i>cat</i> (1)	mkdir	<i>mkdir</i> (1)
cd	<i>cd</i> (1)	mv	<i>cp</i> (1)
cp	<i>cp</i> (1)	pr	<i>pr</i> (1)
cpio	<i>cpio</i> (1)	red	<i>ed</i> (1)
echo	<i>echo</i> (1)	rm	<i>rm</i> (1)
ed	<i>ed</i> (1)	rmdir	<i>rm</i> (1)
expr	<i>expr</i> (1)	sed	<i>sed</i> (1)
find	<i>find</i> (1)	sh	<i>sh</i> (1)
grep	<i>grep</i> (1)	sort	<i>sort</i> (1)
ln	<i>cp</i> (1)	tr	<i>tr</i> (1)
ls	<i>ls</i> (1)	wc	<i>wc</i> (1)

The present issue of the Portability Guide only defines the operation of these commands with respect to 8-bit transparency. It does not define that they will be supplied with suitable localisations of their message text, nor does it guarantee their correct operation with regards to local customs and regular expressions.

The *cc* and *c++* commands will also provide 8-bit transparency for characters contained in string literals. This will enable programmers to define default message strings (see *catgets*(3C)) in languages other than English. The support of 8-bit character constants, 8-bit identifier names and 8-bit characters within comment strings is implementation defined.

It is the intention to add more of the standard commands to this list and to move towards full internationalisation of the above commands.

NAME

`gencat` — generate a formatted message catalogue

SYNOPSIS

`gencat catfile msgfile ....`

DESCRIPTION

Gencat merges the message text source file(s) *msgfile* into a formatted message catalogue *catfile*. *Catfile* will be created if it does not already exist. If *catfile* does exist, its messages will be included in the new *catfile*. If set and message numbers collide, the new message-text defined in *msgfile* will replace the old message text currently contained in *catfile*.

APPLICATION USAGE

Message catalogues produced by *gencat* are binary encoded, meaning that their portability cannot be guaranteed between different types of machine. Thus, just as C programs need to be recompiled for each type of machine, then so message catalogues must be recreated via *gencat*.

CHANGE HISTORY

First released in Issue 2.

Subroutines

This chapter describes the NLS functions conventionally found in the 3C and 3S system libraries. A general description of subroutines and libraries can be found in "XVS SYSTEM CALLS AND LIBRARIES".

Where the same functions are described in this part of the Portability Guide and in "XVS SYSTEM CALLS AND LIBRARIES" (e.g., *isalpha*, *toupper*, *printf*, etc.), then programmers requiring to produce internationalised programs should use the interface definitions described here. Other functions (e.g., *nl_cxtime*, *nl_langinfo*, *nl_strcmp*, etc.) are defined exclusively for the production of internationalised programs and have no counterparts elsewhere in the Portability Guide.

It is an objective that all library functions defined in "XVS SYSTEM CALLS AND LIBRARIES" will provide 8-bit transparency on all conforming systems.

(Note: 8-bit operation of the *curses(3X)* library functions requires further investigation and will be addressed in a future issue.)

NAME

catgetmsg — get message from a message catalogue

SYNOPSIS

```
#include <nl_types.h>
```

```
char *catgetmsg (catd, set_num, msg_num, buf, buflen)  
nl_catd catd;  
int set_num, msg_num, buflen;  
char *buf;
```

DESCRIPTION

Catgetmsg attempts to read up to *buflen*-1 bytes of a message string into the area pointed to by *buf*. *Buflen* is an integer value containing the size in bytes of *buf*. The return string is always terminated with a null byte.

Catd is a catalogue descriptor returned from an earlier call to *catopen*(3C) and identifies the message catalogue containing the message set (*set_num*) and the program message (*msg_num*).

RETURN VALUE

If successful, *catgetmsg* returns a pointer to the message string in *buf*. Otherwise, if *catd* is invalid or *set_num* and/or *msg_num* are not in the message catalogue, *catgetmsg* returns a pointer to an empty (null) string.

SEE ALSO

catopen(3C), *catgets*(3C).

APPLICATION USAGE

Set_num and *msg_num* are defined as integer values for maximum portability. However, it is recommended that programmers should use symbolic names for message and set numbers, wherever possible, rather than having integer values hard-coded into their source programs.

CHANGE HISTORY

First released in Issue 2.

NAME

catgets — read a program message

SYNOPSIS

```
#include <nl_types.h>

char *catgets (catd, set_num, msg_num, s)
nl_catd catd;
int set_num, msg_num;
char *s;
```

DESCRIPTION

Catgets attempts to read message *msg_num*, in set *set_num*, from the message catalogue identified by *catd*. *Catd* is a catalogue descriptor returned from an earlier call to *catopen*(3C). *S* points to a default message string which will be returned by *catgets* if the identified message catalogue is not currently available.

RETURN VALUE

If the identified message is retrieved successfully, *catgets* returns a pointer to an internal buffer area containing the null terminated message string. If the call is unsuccessful because the message catalogue identified by *catd* is not currently available, a pointer to *s* is returned. Otherwise, if the message catalogue is available but the identified message is not contained therein, a pointer to an empty (null) string is returned.

SEE ALSO

catgetmsg(3C), catopen(3C).

APPLICATION USAGE

The message-text is contained in an internal buffer area and should be copied by the application if it is to be saved or re-used after further calls to *catgets*.

Set_num and *msg_num* are defined as integer values for maximum portability. However, it is recommended that programmers should use symbolic names for message and set numbers, wherever possible, rather than having integer values hard-coded into their source programs.

CHANGE HISTORY

First released in Issue 2.

NAME

catopen, catclose — open/close a message catalogue

SYNOPSIS

```
#include <nl_types.h>
```

```
nl_catd catopen (name, oflag)
```

```
char *name;
```

```
int oflag;
```

```
int catclose (catd)
```

```
nl_catd catd;
```

DESCRIPTION

Catopen opens a message catalogue and returns a catalogue descriptor. *Name* specifies the name of the message catalogue to be opened. If *name* contains a "/" then *name* specifies a pathname for the message catalogue. Otherwise, the environment variable *NLSPATH* is used with *name* substituted for %N (see *environ*(5)). If *NLSPATH* does not exist in the environment, or if a message catalogue cannot be opened in any of the paths specified by *NLSPATH*, then an implementation defined default path is used.

Oflag is reserved for future use and should be set to 0 (zero). The results of setting this field to any other value are undefined.

Catclose closes the message catalogue identified by *catd*.

RETURN VALUE

If successful, *catopen* returns a message catalogue descriptor for use on subsequent calls to *catgetmsg*, *catread* and *catclose*. If unsuccessful, *catopen* returns -1.

Catclose returns 0 if successful, otherwise -1.

SEE ALSO

catgetmsg(3C), catgets(3C), environ(5).

APPLICATION USAGE

Using *catopen* may cause another file descriptor to be allocated by the calling process, in addition to *stdin*, *stdout* and *stderr*. Applications should take care not to close this file descriptor by mistake.

CHANGE HISTORY

First released in Issue 2.

NAME

`toupper`, `tolower`, `_toupper`, `_tolower`, `toascii` — translate characters

SYNOPSIS

```
#include <ctype.h>
```

```
int toupper (c)
```

```
int c;
```

```
int tolower (c)
```

```
int c;
```

```
int _toupper (c)
```

```
int c;
```

```
int _tolower (c)
```

```
int (c)
```

```
int toascii (c)
```

```
int c;
```

DESCRIPTION

Toupper and *tolower* have as domain the range of *getc*(3S). If the argument to *toupper* represents a lower-case letter, as defined by the shift tables for the language identified by the last successful call to *nl_init*, the result is the corresponding upper-case letter. If the argument to *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

If *nl_init* has not been called successfully, or if shift information is not available for a supported language, these functions determine the case of characters according to the rules of the ASCII coded character set.

_toupper and *_tolower* are macros that provide the same functionality as *toupper* and *tolower* on valid input, but have restricted domains and are faster. The macro *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. The macro *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

Toascii yields its arguments with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ctype(3C), *getc*(3S), *nl_init*(3C), *ctype*(5).

FUTURE DIRECTIONS

Conv(3C) may be extended to provide character translation functions which convert *c* from one coded character set to another. This will allow character set conversion utilities, for example, to be written in a portable way.

CHANGE HISTORY

First released in Issue 2.

NAME

ctime, *nl_ctime*, *localtime*, *gmtime*, *asctime*, *nl_asctime*, *timezone*, *daylight*, *tzname*, *tzset* — convert date and time to string

SYNOPSIS

```
#include <time.h>

char *ctime (clock)
long *clock;

char *nl_ctime (clock, format)
long *clock;
char *format;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

char *nl_asctime (tm, format)
struct tm *tm;
char *format;

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset();
```

DESCRIPTION

Ctime converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970, and returns a pointer to a 26 character string in the following form:

```
Sun Sep 16 01:03:52 1973 \n \0
```

All the fields have constant width. *Nl_ctime* extends the functionality of *ctime* by allowing date and time information to be output in a number of different ways, as defined by *format*, and by providing month and weekday names in an appropriate form, as defined by the last successful call to *nl_init*.

Format contains a string which is similar to the format strings given as the first argument to *printf*(3S), where each field descriptor is preceded by % and will be replaced in the output by its corresponding value. A single % is coded as %%. All other characters are copied to the output without change.

Field Descriptors:

n insert a newline character
 t insert a tab character
 m month of year - 01 to 12
 d day of month - 01 to 31
 Y last 2 digits of year - 00 to 99
 D date as mm/dd/yy
 H hour - 00 to 23
 M minute - 00 to 59
 S second - 00 to 59
 T time as HH:MM:SS
 j day of year - 001 to 366
 w day of week - 0 to 6 (Sunday = 0)
 a abbreviated weekday - e.g. Sun to Sat
 h abbreviated month - e.g. Jan to Dec
 r time in AM/PM notation

If *format* is the null string, the *D_T_FMT* string defined in the *langinfo* data of the language identified by the last successful call to *nl_init* is used instead. The correct translations of month and weekday names (when selected as alphabetic by the *format* string) are also obtained from *langinfo* data.

For example, if *format* contained

```
%a, %d %h 19%y %H:%M:%S
```

and Italy was identified as the required territory, then on Saturday, May 17, 1986, the date and time would be returned from *nl_cxtime* as:

```
sab, 17 mar 1986 11:46:50
```

If *nl_init* has not been called successfully, or if *langinfo* data is not available for a supported language, and *format* is the null string, then *nl_cxtime* functions like *ctime*; or, if *format* is not the null string, the system default month and weekday names will be used.

Localtime and *gmtime* return pointers to *tm* structures, described below. *Localtime* corrects for time zone and possible Daylight Savings Time. *Gmtime* converts directly to Greenwich Mean Time (GMT), which is the time the system uses.

Asctime converts a *tm* structure to a 26 character string, as shown in the above example of *ctime*, and returns a pointer to the string.

Nl_ascxtime, like *nl_cxtime*, allows the date string to be formatted, and month and weekday names to be presented in an appropriate form. Like *asctime*, it takes a *tm* structure as its argument.

If *nl_init* has not been called successfully, or if *langinfo* data is not available for a supported language, and *format* is the null string, then *nl_ascxtime* functions like *asctime*; or, if *format* is not the null string, the system default month and weekday names will be used.

Declarations of all the functions and externals, and the *tm* structure, are defined in the `<time.h>` header file. The *tm* structure contains the following fields:

```
int tm_sec;      /* seconds (0-59) */
int tm_min;      /* minutes (0-59) */
int tm_hour;     /* hours (0-23) */
int tm_mday;     /* day of month (1-31) */
int tm_mon;      /* month of year (0-11) */
int tm_year;     /* year — 1900 */
int tm_wday;     /* day of week (Sunday = 0) */
int tm_yday;     /* day of year (0 - 365) */
int tm_isdst;    /* Daylight Savings Time */
```

Tm_isdst is non-zero if Daylight Savings Time is in effect.

The external variable *timezone* contains the difference in seconds between GMT and the local standard time (in MET, *timezone* is $-1*60*60$); the external variable *daylight* is non-zero if (and only if) some Daylight Savings Time conversion should be applied.

If an environment variable TZ is present, *asctime* and *nl_asctime* use the contents of the variable to override the default time zone. The value of TZ must be a three letter time zone name, followed by an optional minus sign (for time zones east of Greenwich) and a series of digits representing the difference between local time and GMT in hours. This is followed by an optional three letter name for a daylight time zone. The setting for most of continental Europe would be *MET-1EET*. The effects of setting TZ are thus to change the values of the external variables *timezone* and *daylight*. In addition, the time zone names contained in the external variable

```
char *tzname[2] = ("MET" "EET");
```

are set from the environment variable TZ. The function *tzset* sets these external variables from TZ. *Tzset* is called by *asctime* and *nl_asctime* and may also be called directly by the user.

Note that Daylight Savings Time conversion is applied all year round if there is a daylight time zone name present in the TZ variable, i.e. *daylight* is non-zero all year round, but *tm_isdst* is non-zero only when Daylight Saving Time is in effect.

SEE ALSO

time(2), *getenv(3C)*, *nl_init(3C)*, *printf(3S)*, *environ(5)*, *langinfo(5)*.

CHANGE HISTORY

First released in Issue 2.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii — classify characters

SYNOPSIS

```
#include <ctype.h>
```

```
int isalpha(c)
```

```
int c;
```

```
...
```

DESCRIPTION

These macros classify character-coded integer values according to the rules of the coded character set identified by the last successful call to *nl_init*. Each macro is a predicate returning non-zero for true, zero for false.

If *nl_init* has not been called successfully, or if character classification information is not available for a supported language, then characters are classified according to the rules of the ASCII 7-bit coded character set, returning 0 for values above octal 0177.

isascii is defined on all integer values. The rest are defined on EOF and values in the character range of the codeset identified by the last successful call to *nl_init*.

isalpha

c is a letter (upper-case or lower-case).

isupper

c is an upper-case letter.

islower

c is a lower-case letter.

isdigit

c is a digit (e.g. the characters [0-9] in ASCII).

isxdigit

c is a hexadecimal digit (e.g. the characters [0-9], [A-F] or [a-f] in ASCII).

isalnum

c is an alphanumeric (letter or digit).

isspace

c is a character causing "white space" in displayed text (e.g. space, tab, carriage return, newline, vertical tab or form-feed in ASCII).

ispunct

c is a punctuation character (e.g. any printing character except the space character (040), digits or letters in ASCII).

isprint

c is a printing character.

isgraph

c is a character with a visible representation (e.g. printing characters excluding the space character (040) in ASCII).

iscntrl

c is a control character (e.g. character codes less than 040 and the delete character (0177) in ASCII).

isascii

c is an ASCII character, code between 0 and 0177 inclusive.

RETURN VALUE

If the argument to any of these functions is not in the domain of the function, then the result is undefined.

SEE ALSO

nl_init(3C), stdio(3S), ctype(5).

FUTURE DIRECTIONS

Additional functions may be added to check character validity within coded character sets other than ASCII, and to provide character classification for non-spacing characters - e.g. non-spacing blanks in the ISO 8859/1 coded character set, and non-spacing diacritical marks in ISO 6937.

CHANGE HISTORY

First released in Issue 2.

NAME

ecvt, *fcvt*, *gcv*t — convert floating-point number to string

SYNOPSIS

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcv
```

DESCRIPTION

Ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer thereto. The high-order digit is non-zero, unless the *value* is zero (0). The low-order digit is rounded. The position of the radix character relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The radix character is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero. The symbol used to represent a radix character can be determined by calling:

```
nl_langinfo (RADIXCHAR);
```

*Fcv*t is identical to *ecvt*, except that the correct digit has been rounded for *printf*(3S) *%f* (FORTRAN F-format) output of the number of digits specified by *ndigit*.

*Gcv*t converts *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F-format (*[-]ddd.ddd*) if possible, otherwise E-format (*[-]ddde±dd*), ready for printing. A minus sign, if there is one, or a radix character will be included as part of the return string. The radix character is that defined for the language identified by the last successful call to *nl_init*. Trailing zeros are suppressed.

If *nl_init* has not been called successfully, or if the radix character is not defined for a supported language, the radix character defaults to a period (.

SEE ALSO

nl_init(3C), *nl_langinfo*(3C), *printf*(3S).

CHANGE HISTORY

First released in Issue 2.

NAME

nl_init - initialise NLS operation

SYNOPSIS

```
int nl_init (lang)
char *lang;
```

DESCRIPTION

Nl_init initialises NLS operation for the language identified by *lang*, which is a pointer to a string containing settings of *language*, *territory* and *codeset* as defined for the *LANG* environment variable (see *environ(5)*).

Typically, *nl_init* is used to bind program operation to the user's specified language requirements, e.g.

```
nl_init (getenv ("LANG"));
```

A call to *nl_init* will fail if the string pointed to by *lang* does not identify a valid *language/territory/codeset* combination. If *nl_init* has already been called successfully, operation will continue for the language identified on the last successful call. Otherwise, if *nl_init* has not already been called successfully, the various NLS library functions will each revert to their own default mode of operation.

RETURN VALUE

If successful, *nl_init* will return 0 (zero). Otherwise, -1 will be returned.

SEE ALSO

getenv(3C), *environ(5)*.

APPLICATION USAGE

An internationalised application program will not provide the correct language operation until *nl_init* has been called successfully. Subsequent calls to *nl_init* can be used to switch operation from one supported language to another.

CHANGE HISTORY

First released in Issue 2.

NAME

nl_langinfo — language information

SYNOPSIS

```
#include <nl_types.h>
```

```
#include <langinfo.h>
```

```
char *nl_langinfo (item)
```

```
nl_item item;
```

DESCRIPTION

Nl_langinfo returns a pointer to a null-terminated string containing information relevant to a particular language or cultural area identified by the last successful call to *nl_init*. The manifest constant names and values of *item* are defined in *langinfo.h*. For example:

```
nl_langinfo (ABDAY_1);
```

would return a pointer to the string "Dom" if the identified language was Portuguese, and "Sun" if the identified language was English.

If *nl_init* has not been called successfully, or if *langinfo* data for a supported language is either not available or *item* is not defined therein, then *nl_langinfo* returns a pointer to an empty (null) string.

SEE ALSO

nl_init(3C), environ(5), langinfo(5), nl_types(5).

CHANGE HISTORY

First released in Issue 2.

NAME

printf, fprintf, sprintf, nl_printf, nl_fprintf, nl_sprintf — print formatted output

SYNOPSIS

```
int printf (format [, arg ...])
char *format;

int fprintf (stream, format [, arg ...])
FILE *stream;
char *format;

int sprintf (s, format [, arg ...])
char *s, *format;

int nl_printf (format [, arg ...])
char *format;

int nl_fprintf (stream, format [, arg ...])
FILE *stream;
char *format;

int nl_sprintf (s, format [, arg ...])
char *s, *format;
```

DESCRIPTION

Printf places output on the standard output stream `stdout`. *Fprintf* places output on the named output stream. *Ssprintf* places output followed by the NULL character (`\0`), in consecutive bytes starting at `*s`; it is the user's responsibility to ensure that enough space is available. Each function returns the number of characters transmitted (not including the `\0` in the case of *ssprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats and prints its *args* under the control of *format*. The *format* is a character string that contains two types of objects; plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in the fetching of zero or more *args*. The results are undefined if there are insufficient *args* for the *format*. If the *format* is exhausted while *args* remain, the excess *args* are simply ignored.

The NLS functions *nl_printf*, *nl_fprintf* and *nl_sprintf* provide similar functionality, with the difference that the conversion character `%` in *printf* (see below) is replaced by the sequence `%digit$`, where *digit* is a decimal digit *n* in the range (1-`{NL_ARGMAX}`). Conversions are applied to the *n*th argument in the argument list, rather than to the next unused argument.

The *format* passed to the NLS printing functions can contain either form of a conversion specification, i.e. `%` or `%digit$`, although the two forms cannot be mixed within a single *format* string.

Printf formatting in all its forms provides for the insertion of a language dependent radix character in the output string. The radix character inserted is that defined for the language identified by the last successful call to *nl_init*. If *nl_init* has not been called successfully, or if the radix character is not defined for a supported language, the radix character is defined as a period (.).

Each conversion specification is introduced by either the % character (in the *printf* case) or by the character sequence *%digit\$*, after which the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left adjustment flag —, described below, has been given) to the field width.

A *precision* that gives the minimum number of digits to appear for the *d*, *o*, *u*, *x*, or *X* conversions, the number of digits to appear after the radix character for the *e* and *f* conversions, the maximum number of significant digits for the *g* conversion, or the maximum number of characters to be printed from a string in the *s* conversion. The precision takes the form of a period (.) followed by a decimal digit string, where a null digit string is treated as zero.

An optional *l* (ell) specifying that a following *d*, *o*, *u*, *x* or *X* conversion character applies to a long integer *arg*. An *l* before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk '*' instead of a digit string in *format* strings containing the % form of a conversion specification. In this case, an integer *arg* supplies the field width or precision. The results of specifying an asterisk '*' in *format* strings containing *%digit\$* conversion specifications are undefined.

The flag characters and their meanings are:

- The result of the conversion will be left-justified within the field.
- + The result of a signed conversion will always begin with a sign (+ or —).

blank

If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

- # This flag specifies that the value is to be converted to an "alternate form". For *c*, *d*, *s* and *u* conversions, the flag has no effect. For *o* conversion, it increases the precision to force the first digit of the result to be zero. For *x* and *X* conversion, a non-zero result will have *Ox* or *OX* prefixed to it. For *e*, *E*, *f*, *g* or *G* conversions, the result will always contain a radix character even if no digits follow the radix character (normally, a radix character appears in the result of these conversions only if a digit follows it). For *g* and *G* conversions, trailing zeros will not be removed from the result as they normally are.

The conversion characters and their meanings are:

d, o, u, x, X

The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (*x* and *X*), respectively. The letters *abcdef* are used for *x* conversion and the letters *ABCDEF* for *X* conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

f The float or double *arg* is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the radix character is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no radix character appears.

e, E

The float or double *arg* is converted in the style *[]d.ddde±dd*, where there is one digit before the radix character and the number of digits after the radix character is equal to the precision. When the precision is missing, six digits are produced. If the precision is zero, no radix character appears. The *E* format code will produce a number with *E* instead of *e* introducing the exponent. The exponent always contains at least two digits. However, if the value to be printed is greater than or equal to $1E+100$, additional exponent digits will be printed as necessary.

g, G

The float or double *arg* is printed in style *f* or *e* (or in style *E* in the case of a *G* format code), with the precision specifying the number of significant digits. The style used depends on the value converted; style *e* will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result; a radix character appears only if it is followed by a digit.

c The character *arg* is printed.

s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a NULL (`\0`) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.

% Print a *%*; no argument is converted.

If the character after the *%* or *%digit\$* sequence is not a valid conversion character, the results of the conversion are not predictable.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf*, *fprintf*, *nl_printf* and *nl_fprintf* are printed as if *putc(3S)* had been called.

SEE ALSO

`nl_init(3C)`, `putc(3S)`, `scanf(3C)`, `stdio(3S)`, `langinfo(5)`.

APPLICATION USAGE

The effects of using the same value of *digit* more than once in the same *format* string are undefined. Some implementations may permit this to happen, and produce the expected results, while others may regard it as an error. For maximum portability, therefore, it is recommended that each argument should be used exactly once and that the number of arguments should be equal to the number of substitutions specified in the *format* string.

EXAMPLES

To print the language independent date and time format using the following `nl_printf` statement:

```
nl_printf (format, weekday, month, day, hour, min);
```

For American usage, *format* could be a pointer to the string:

```
"%1$s, %2$s %3$d, %4$d:%5$.2d\n"
```

producing the message:

```
Sunday, July 3, 10:02
```

whereas for German usage, *format* could be a pointer to the string:

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

producing the message:

```
Sonntag, 3. Juli, 10:02
```

CHANGE HISTORY

First released in Issue 2.

NAME

scanf, fscanf, sscanf, nl_scanf, nl_fscanf, nl_sscanf — convert formatted input

SYNOPSIS

```
#include <stdio.h>

int scanf (format [, pointer ...])
char *format;

int fscanf (stream, format [, pointer ...])
FILE *stream;
char *format;

int sscanf (s, format [, pointer ...])
char *s, *format;

int nl_scanf (format [, pointer ...])
char *format;

int nl_fscanf (stream, format [, pointer ...])
FILE *stream;
char *format;

int nl_sscanf (s, format [, pointer ...])
char *s, *format;
```

DESCRIPTION

Scanf reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a specified format, and stores the results in its arguments. Each expects, as arguments, a control string *format* (described below), and a set of *pointer* arguments indicating where the converted input should be stored.

The NLS functions *nl_scanf*, *nl_fscanf* and *nl_sscanf* provide similar functionality with the difference that the conversion character % in *scanf* (see below) is replaced by the sequence *%digit\$*, where *digit* is a decimal digit *n* in the range (1-*{NL_ARGMAX}*). Conversions are applied to the *n*th argument in the argument list, rather than to the next unused argument.

The *format* passed to the NLS functions can contain either form of a conversion specification, i.e. % or *%digit\$*, although the two forms cannot be mixed within a single *format* string.

Scanf in all its forms provides for the detection of a language dependent radix character in the input string. The radix character used is that defined for the language identified by the last successful call to *nl_init*. If *nl_init* has not been called successfully, or if the radix character is not defined for a supported language, the radix character is defined as a period (.).

The *format* string usually contains conversion specifications, which are used to direct interpretation of input character sequences. The *format* string can contain:

1. White-space characters (blanks, tabs, new-lines or form-feeds), which, except in the two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of either the character % or the character sequence *%digit\$*, an optional assignment suppressing character *, an optional numeric maximum field width, and optional *l* (ell) or *h* indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field, where the result is stored in the variable pointed to by the corresponding argument in the argument list, unless assignment suppression is indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters, which extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except *l* and *c*, white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field, where the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

- % a single % is expected in the input at this point; no assignment is done.
- d* a decimal integer is expected; the corresponding argument should be an integer pointer.
- u* an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.
- o* an octal integer is expected; the corresponding argument should be an integer pointer.
- x* a hexadecimal number is expected; the corresponding argument should be an integer pointer.
- e, f, g* a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a radix character, followed by an optional exponent field consisting of an *E* or *e*, followed by an optional +, —, or space, followed by an integer.

- s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating `\0`, which will be added automatically. The input field is terminated by a white-space character.
- c a character is expected; the corresponding argument should be a character pointer. The normal skip over white-space is suppressed in this case; to read the next non-space character, use either `%1s` or `%n$1s`. If a field width is given, the corresponding argument should refer to a character array, into which the indicated number of characters will be read.
- [indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, called the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters not contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first-last*, thus `[0123456789]` may be expressed as `[0-9]`. Using this convention, *first* must be lexically less than or equal to *last* in the machine collating sequence, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or last character in a scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating `\0`, which will be added automatically. At least one character must match for this conversion to be considered successful.

If an invalid conversion character follows a `%` or `%digit$` sequence, the results of the operation may not be predictable.

The conversion characters *d*, *u*, *o* and *x* may be preceded by an *l* or *h* to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters *e*, *f* and *g* may be preceded by *l* to indicate that a pointer to **double** rather than to **float** is in the argument list. The *l* or *h* modifier is ignored for other conversion characters.

Scanf and *nl_scanf* conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Scanf and *nl_scanf* return the number of successfully matched and assigned input items. This number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

RETURN VALUE

These functions return EOF on end of input and a short count for missing or illegal data items.

SEE ALSO

getc(3S), nl_init(3C), printf(3C), strtod(3C), strtol(3C), langinfo(5).

APPLICATION USAGE

Use of the "range" facility within a scanset is likely to make the program language and/or codeset dependent, and should be avoided.

FUTURE DIRECTIONS

When an international version of *regcmp*(3X) is defined, it is likely that the NLS conversion functions will be updated to provide operators that match characters in terms of language collating sequences.

CHANGE HISTORY

First released in Issue 2.

NAME

strcat, strncat, strcmp, strncmp, nl_strcmp, nl_strncmp, strcpy, strncpy, strlen, strchr, strchr, strpbrk, strspn, strtok — string operations

SYNOPSIS

```
#include <string.h>
```

```
char *strcat (s1, s2)
```

```
char *s1, *s2;
```

```
char *strncat (s1, s2, n)
```

```
char *s1, *s2;
```

```
int n;
```

```
char *strcmp (s1, s2)
```

```
char *s1, *s2;
```

```
char *strncmp (s1, s2, n)
```

```
char *s1, *s2;
```

```
int n;
```

```
int nl_strcmp (s1, s2)
```

```
char *s1, *s2;
```

```
int nl_strncmp (s1, s2, n)
```

```
char *s1, *s2;
```

```
int n;
```

```
char *strcpy (s1, s2)
```

```
char *s1, *s2;
```

```
char *strncpy (s1, s2, n)
```

```
char *s1, *s2;
```

```
int n;
```

```
int strlen (s)
```

```
char *s;
```

```
char *strchr (s, c)
```

```
char *s;
```

```
int c;
```

```
char *strrchr (s, c)
```

```
char *s;
```

```
int c;
```

```
char *strpbrk (s1, s2)
```

```
char *s1, *s2;
```

```
int strspn (s1, s2)
```

```
char *s1, s2;
```

```
int strcspn (s1, s2)
```

```
char *s1, s2;
```

```
int strtok (s1, s2)
char *s1, s2;
```

DESCRIPTION

The arguments *s1*, *s2* and *s* point to strings (arrays of characters terminated by a NULL character). The functions *strcat*, *strncat*, *strcpy* and *strncpy* all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

Strcat appends a copy of string *s2* to the end of string *s1*. *Strncat* appends at most *n* characters. Each returns a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer less than, equal to, or greater than zero (0), according to whether *s1* is lexicographically less than, equal to, or greater than *s2* in the ASCII collating sequence. *Strncmp* makes the same comparison but looks at *n* characters at most.

The NLS functions *nl_strcmp* and *nl_strncmp* provide similar facilities, using the language dependent collating information defined for the language identified by the last successful call to *nl_init*. If *nl_init* has not been called successfully, or if collating information is not available for a supported language, then *nl_strcmp* and *nl_strncmp* work like *strcmp* and *strncmp*.

Nl_strcmp and *nl_strncmp* compare strings according to the language dependent "dictionary" ordering of data, i.e. taking into account case and accent priority, "1-to-1", "1-to-2", "2-to-1" and "don't care" character mappings.

Strcpy copies string *s2* to *s1*, stopping after the NULL character has been copied. *Strncpy* copies exactly *n* characters, truncating *s2* or adding NULL characters to *s1* if necessary. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

Strlen returns the number of characters in *s*, not including the terminating NULL character.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or a NULL pointer if *c* does not occur in the string. The NULL character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a NULL pointer if no character from *s2* exists in *s1*.

Strspn (*strcspn*) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

Strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of the position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string *s1* immediately following that token. In this way subsequent calls will work through string *s1*, returning a pointer to the first character of each subsequent token. A NULL character will have been written into *s1* by *strtok*

immediately following the token. The separator string *s2* may be different from call to call. When no tokens remain in *s1*, a NULL pointer is returned.

SEE ALSO

memory(3C).

APPLICATION USAGE

Most of these functions are declared in the `<string.h>` header file.

Strcmp and *strncmp* use binary character comparison. The sign of the value returned when one of the characters has its high-order bit set is implementation dependent.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

FUTURE DIRECTIONS

The type of the argument *n* to *strncat*, *strncmp*, *strncpy* and *nl_strncmp*, and the type of the value returned by *strlen*, will be declared through the *typedef* facility in a header file as *size_t*.

Currently there is no reliable way of informing the caller that collating sequence information is not available for a supported language. When this occurs, the wrong results can be returned from *nl_strcmp* or *nl_strncmp*. For this reason, it may be necessary to provide extra error checking functions in future releases of the interface definition.

Nl_strcmp and *nl_strncmp* provide for the "dictionary" ordering of data. It may also be necessary to provide functions which compare strings in other than "dictionary" order. The names *nl_chrcmp* and *nl_chrncmp* are reserved for this purpose.

CHANGE HISTORY

First released in Issue 2.

NAME

strtod, atof — convert string to double precision number

SYNOPSIS

```
double strtod (str, ptr)
```

```
char *str, **ptr;
```

```
double atof (str)
```

```
char *str;
```

DESCRIPTION

Strtod returns, as a double-precision floating-point number, the value represented by the character *string* pointed to by *str*. The string is scanned up to the first unrecognised character.

Strtod recognises an optional string of “white-space” characters (as defined by *isspace* in *ctype*(3C)), then an optional sign, then a string of digits optionally containing a language dependent radix character, then an optional *e* or *E* followed by an optional sign, followed by an integer.

The radix character is that defined for the language identified by the last successful call to *nl_init*. If *nl_init* has not been called successfully, or if the radix character is not defined for a supported language, the radix character is defined as a period (.).

If the value of *ptr* is not (*char ***) NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, **ptr* is set to *str* and zero is returned.

Atof (str)

is equivalent to

```
strtod (str, (char **) NULL);
```

RETURN VALUE

If the correct value would cause overflow, $\pm$ HUGE is returned (according to the sign of the value), and *errno* is set to [ERANGE].

If the correct value would cause underflow, zero is returned and *errno* is set to [ERANGE].

SEE ALSO

ctype(3C), *nl_init*(3C), *scanf*(3S), *strtol*(3C), *langinfo*(5).

CHANGE HISTORY

First released in Issue 2.

Header Files

This chapter describes the NLS header files. For a complete description of header files and their usage see "XVS SYSTEM CALLS AND LIBRARIES".

NAME

ctype — NLS character types

SYNOPSIS

```
#include <ctype.h>
```

DESCRIPTION

The following macros are defined in this header file:

isalpha(c)

c is a letter (upper-case or lower-case).

isupper(c)

c is an upper-case letter.

islower(c)

c is a lower-case letter.

isdigit(c)

c is a decimal digit (e.g. the characters [0-9] in ASCII).

isxdigit(c)

c is a hexadecimal digit (e.g. the characters [0-9], [A-F] or [a-f] in ASCII).

isalnum(c)

c is an alphanumeric (letter or digit).

isspace(c)

c is a character causing "white space" in displayed text (e.g. space, tab, carriage return, new-line, vertical tab and form-feed in ASCII).

ispunct(c)

c is a punctuation character (e.g. any printing character except the space character (040), digits or letters in ASCII).

isprint(c)

c is a printing character.

isgraph(c)

c is a character with a visible representation (e.g. printing characters excluding the space character in ASCII).

iscntrl(c)

c is a control character (e.g. character codes less than 040 and the delete character (0177) in ASCII).

isascii(c)

c is an ASCII character, code between 0 and 0177 inclusive.

_toupper(c)

converts the lower-case letter *c* to the corresponding upper-case letter

_tolower(c)

converts the upper-case letter *c* to the corresponding lower-case letter

toascii(c)

converts *c* to an ASCII character by masking out high-order bits

SEE ALSO

nl_init(3C).

APPLICATION USAGE

_toupper and *_tolower* both give undefined results if their arguments are not of the correct lower or upper case respectively.

It is not mandatory that the above interface routines are supplied as macros.

CHANGE HISTORY

First released in Issue 2.

NAME

environ - NLS environment variables

DESCRIPTION

The following environment variables are defined by the X/OPEN NLS interface. and are additional to those defined in "XVS SYSTEM CALLS AND LIBRARIES", *environ*(5). These variables are made available to a process by *exec*(2).

LANG

Identifies the user's requirements for native language, local customs and coded character set, in the form

```
LANG=language[_territory[.codeset]]
```

Values of *LANG* are given in English as an ASCII character string. At runtime, the user's language requirements, as specified by the setting of *LANG*, are bound to the execution of a program by calling *nl_init*, e.g.

```
nl_init (getenv ("LANG"));
```

LANG can be defined on a system-wide basis (e.g. in */etc/profile*), or for each user individually (e.g. in *\$HOME/.profile*). The value of *LANG* can also be changed dynamically by interactive users without requiring that the command interpreter is re-started. For example, the command sequence:

```
$ LANG=spanish
$ export LANG
```

changes the value of *LANG* with immediate effect.

NLSPATH

Contains a sequence of pseudo-pathnames which *catopen* uses when attempting to locate message catalogues. Each pseudo-pathname contains a name template, consisting of an optional path-prefix, one or more substitution fields, a filename and an optional filename suffix.

For example:

```
NLSPATH="/system/nlslib/%N.cat"
```

defines that *catopen* should look for all message catalogues in the directory */system/nlslib*, where the catalogue name should be constructed from the *name* parameter passed to *catopen*(%N), with the suffix *.cat*.

Substitution fields consist of a % symbol, followed by a single-letter keyword. The following keywords are currently defined:

%N The value of the *name* parameter passed to *catopen*.

%L The value of *LANG*.

%l The *language* element from *LANG*.

%t The *territory* element from *LANG*.

`%c` The *codeset* element from *LANG*.

`%%` A single `%` character.

A null string is substituted if the specified value is not currently defined. The separators `"_"` and `"."` are not included in `%t` and `%c` substitutions.

Pathnames defined in *NLSPATH* are separated by colons (`:`). A leading or two adjacent colons (`::`) indicates the current directory. For example:

```
NLSPATH=" :%N.cat : /nlslib/%L/%N.cat"
```

indicates to *catopen* that it should look for the requested message catalogue in *name*, *name.cat* and */nlslib/\$LANG/name.cat*.

SEE ALSO

`exec(2)`, `catopen(3C)`, `nl_init(3C)`.

`environ(5)` in "XVS SYSTEM CALLS AND LIBRARIES".

FUTURE DIRECTIONS

As currently defined, *LANG* identifies the natural language requirements for both the internal processing of an application and for the presentation of program messages. This may be limiting if, for example, a user requires to interface to an application in French while processing German language text files. The interface definition is likely to be enhanced to separate these two cases. (Note that the environment variable name *PROCLANG* is reserved so that it can be used for this purpose.)

CHANGE HISTORY

First released in Issue 2.

NAME

langinfo — language information constants

SYNOPSIS

```
#include <langinfo.h>
```

DESCRIPTION

This header file contains the constants used to identify items of langinfo data (see *nl_langinfo(3C)*). The mode of *items* is given in *nl_types(5)*. At least the following constants will be defined on all X/OPEN systems:

<i>D_T_FMT</i>	string for formatting date and time
<i>DAY_1</i>	name of the first day of the week (i.e. Sunday)
<i>DAY_7</i>	name of the seventh day of the week
<i>ABDAY_1</i>	abbreviated name of the first day of the week
<i>ABDAY_7</i>	abbreviated name of the seventh day of the week
<i>MON_1</i>	name of the first month in the Gregorian calendar
<i>MON_12</i>	name of the twelfth month
<i>ABMON_1</i>	abbreviated name of the first month
<i>ABMON_12</i>	abbreviated name of the twelfth month
<i>RADIXCHAR</i>	radix character
<i>THOUSEP</i>	separator for thousands
<i>YESSTR</i>	affirmative response for yes/no queries
<i>NOSTR</i>	negative response for yes/no queries
<i>CRNCYSTR</i>	currency symbol, preceded by — if the symbol should appear before the value, or + if the symbol should appear after the value

SEE ALSO

nl_langinfo(3C), *nl_types(5)*.

CHANGE HISTORY

First released in Issue 2.

NAME

limits - implementation specific constants

SYNOPSIS

```
#include <limits.h>
```

DESCRIPTION

The following constants are specific to NLS operation and are additional to those specified in "XVS SYSTEM CALLS AND LIBRARIES".

Name	Description	Portability Value
NL_ARGMAX	max value of "digit" in calls to the NLS printf and scanf functions	9
NL_MSGMAX	max message number	32767
NL_SETMAX	max set number	255
NL_TEXTMAX	max no. of bytes in a message string	255

SEE ALSO

limits(5) in "XVS SYSTEM CALLS AND LIBRARIES".

CHANGE HISTORY

First released in Issue 2.

NAME

nl_types - NLS data types

SYNOPSIS

```
#include <nl_types.h>
```

DESCRIPTION

Some data types used in the NLS interface definition are implementation dependent. These are defined in the `<nl_types.h>` header file, which contains definitions of at least the following types:

nl_catd

used by the message catalogue functions *catopen*, *catgetmsg*, *catgets* and *catclose* to identify a catalogue descriptor.

nl_item

used by *nl_langinfo(3C)* to identify items of *langinfo* data.

Values of *nl_catd* are implementation defined. Values of *nl_item* are defined in `<langinfo.h>`.

SEE ALSO

catgetmsg(3C), *catgets(3C)*, *catopen(3C)*, *nl_langinfo(3C)*, *langinfo(5)*.

CHANGE HISTORY

First released in Issue 2.

X/O/P/E/N

PORTABILITY GUIDE

THE X/OPEN SYSTEM V SPECIFICATION
TERMINAL INTERFACES

Contents

Chapter	1	INTRODUCTION
Chapter	2	CURSES OVERVIEW
Chapter	3	DATA TYPES AND THE CURSES.H HEADER FILE
	3.1	THE < curses.h > HEADER FILE
	3.2	DATA TYPES
	3.3	CONSTANTS
	3.3.1	General
	3.3.2	Video Attributes
	3.3.3	Input Values
Chapter	4	FUNCTIONS
	4.1	FUNCTION CATEGORIES
	4.2	NAMING CONVENTIONS
	4.3	EXAMPLE OF USE
Chapter	5	SYNCHRONOUS AND NETWORKED ASYNCHRONOUS TERMINALS
	5.1	OUTPUT
	5.2	INPUT
Chapter	6	CALL SPECIFICATIONS
	6.1	RETURN VALUE
	6.2	< curses.h > HEADER FILE

addch(CURSES)
addstr(CURSES)
attroff(CURSES)
baudrate(CURSES)
beep(CURSES)
box(CURSES)
cbreak(CURSES)
clear(CURSES)
clearok(CURSES)
clrtoebot(CURSES)
clrtoeol(CURSES)
def_prog_mode(CURSES)
delay_output(CURSES)
delch(CURSES)
deleteln(CURSES)
delwin(CURSES)

echo(CURSES)
endwin(CURSES)
erase(CURSES)
erasechar(CURSES)
flushinp(CURSES)
getch(CURSES)
getstr(CURSES)
getyx(CURSES)
has_ic(CURSES)
has_il(CURSES)
idlok(CURSES)
inch(CURSES)
initscr(CURSES)
insch(CURSES)
insertln(CURSES)
intrflush(CURSES)
keypad(CURSES)
killchar(CURSES)
leaveok(CURSES)
longname(CURSES)
move(CURSES)
mvwin(CURSES)
newpad(CURSES)
newterm(CURSES)
newwin(CURSES)
nl(CURSES)
nodelay(CURSES)
overlay(CURSES)
prefresh(CURSES)
printw(CURSES)
reset_prog_mode(CURSES)
raw(CURSES)
refresh(CURSES)
resetty(CURSES)
scanw(CURSES)
scroll(CURSES)
scrollok(CURSES)
set_term(CURSES)
setscrreg(CURSES)
subwin(CURSES)
touchwin(CURSES)
typeahead(CURSES)
unctrl(CURSES)
wnoutrefresh(CURSES)

Introduction

The X/OPEN Group has identified a strong need for a generic terminal interface for applications that are independent of terminal hardware and connection method.

This interface should allow the attachment of character and block oriented terminals. Furthermore, it should not put any constraints on how the terminals are attached (e.g., Local Area Networks, PADs on X.25 etc.).

As a first step towards providing this generic interface, X/OPEN defines the *curses* library routines for general terminal handling. The *curses* library interfaces provide the user with a method of updating screens with reasonable optimisation.

The X/OPEN group has found it impossible to define a totally portable set of *curses* interface routines that cover asynchronous, networked and synchronous terminals.

The functions are oriented towards locally connected asynchronous terminals. For such terminals, applications conforming to this interface are portable. *Curses* may, however, also be used with synchronous and networked asynchronous terminals, provided the restrictions described in "Synchronous and Networked Asynchronous Terminals" are considered.

These functions have been included in the X/OPEN definition in the "optional" category. This means that although they are likely to appear on many X/OPEN systems, they are not guaranteed to be on all. Where they are supported, they will conform to the given definition.

It is envisaged that additional routines will be defined in the future to make better use of the capabilities offered by synchronous terminals.

The X/OPEN Terminal Interfaces Definition is structured as follows:

- | | |
|-----------|--|
| Chapter 2 | gives an overview of the <i>curses</i> interfaces. |
| Chapter 3 | describes the data types used and the <code><curses.h></code> header file. |
| Chapter 4 | explains the broad categories the routines fall into and gives an example of use. |
| Chapter 5 | provides some notes regarding the use of Synchronous and Networked Terminals. |
| Chapter 6 | contains detailed specifications of the X/OPEN definition of <i>curses</i> interfaces. |

Curses Overview

The X/OPEN Terminal Interfaces Definition describes a set of C-language functions that provide screen handling and updating, collectively known as the *curses* library.

The *curses* library permits manipulation of data structures called *windows* which may be thought of as two-dimensional arrays of characters representing all or part of a terminal's screen. The windows are manipulated using a procedural interface described in this section. The *curses* package maintains a record of what characters are on the physical screen. At the most basic level, manipulation is done with the routines *move()* and *addch()* which are used to "move" around and add characters to the default window, *stdscr*, representing the whole screen.

An application may use these routines to add data to the window in any convenient order. Once all data has been added, the routine *refresh()* is called. The package then determines what changes have been made which affect the physical screen. The screen contents are then changed to reflect those characters now in the window using a sequence of operations optimised for the type of terminal in use.

At a higher level routines combining the actions of *move()* and *addch()* are defined, as are routines to add whole strings and to perform format conversions in the manner of *printf(3S)*.

Interfaces are also defined to erase the entire window and to specify the attributes of individual characters in the window. Attributes such as inverse video, underline and blink can be used on a per character basis.

It is possible to create new windows, thus allowing the application to build several images of the screen and display the appropriate one very quickly. New windows are created using the routine *newwin()*. For each routine which manipulates the default window, *stdscr*, there is a corresponding routine named *w...()* to manipulate the contents of a specified window; for example, *move()* and *wmove()*. In fact, *move(...)* is functionally equivalent to *wmove(stdscr, ...)*. This is similar to the *stdio(5)* interface offered by *printf(...)* and *fprintf(stdout, ...)*.

Windows do not have to correspond to the entire screen. It is possible to create smaller windows, and also to indicate that a window is only partially visible on the screen. Furthermore, large windows, or *pads*, which are bigger than the actual screen size, may be created.

The routine *newterm()* may be called to "open" additional terminals by large applications wishing to manipulate several terminals at once. *Set_term()* is used to select the terminal whose screen is to be updated by the next *refresh()*.

Interfaces are also defined to allow input character manipulation and to disable and enable many input attributes: echo, single character input with or without signal processing (*cbreak* or *raw* modes), whether carriage returns are mapped to newlines, whether the screen may be scrolled, etc. .

Data Types and the *curses.h* Header File

The data types supported by *curses* are described in this chapter.

As the library supports a procedural interface to the data types, actual structure contents are not described. All *curses* data is manipulated using the routines provided.

3.1 THE *<curses.h>* HEADER FILE

The *<curses.h>* header file defines various constants and declares the data types which are available to the application.

3.2 DATA TYPES

The following data types are declared:

<i>WINDOW</i> *	pointer to screen representation
<i>SCREEN</i> *	pointer to terminal descriptor
<i>bool</i>	boolean data type
<i>chtype</i>	representation of a character in a window

The actual *WINDOW* and *SCREEN* objects used to store information are created by the corresponding routines and a pointer to them is provided. All manipulation is through that pointer.

3.3 CONSTANTS

The following constants are defined:

3.3.1 General

<i>COLS</i>	number of columns on terminal screen
<i>ERR</i>	value returned on error condition
<i>FALSE</i>	boolean <i>false</i> value
<i>LINES</i>	number of lines on terminal screen
<i>OK</i>	value returned on successful completion
<i>NULL</i>	zero pointer value
<i>TRUE</i>	boolean <i>true</i> value

3.3.2 Video Attributes

These constants may be passed to the routines *attron()*, *attroff()* and *attrset()*:

<i>A_BLINK</i>	blinking
<i>A_BOLD</i>	extra bright or bold
<i>A_DIM</i>	half bright
<i>A_REVERSE</i>	reverse video
<i>A_STANDOUT</i>	terminal's best highlighting mode
<i>A_UNDERLINE</i>	underlining
<i>A_ATTRIBUTES</i>	bit-mask to extract attributes
<i>A_CHARTEXT</i>	bit-mask to extract a character

Normally, attributes are a property of the character.

3.3.3 Input Values

The following constants might be returned by *getch()* if *keypad()* has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or the definition for the key is not present in the underlying table of terminal capabilities.

<i>KEY_BREAK</i>	break key
<i>KEY_DOWN</i>	the four arrow keys ...
<i>KEY_UP</i>	...
<i>KEY_LEFT</i>	...
<i>KEY_RIGHT</i>	...
<i>KEY_HOME</i>	home key (upward+left arrow)
<i>KEY_BACKSPACE</i>	backspace
<i>KEY_F0</i>	function keys; space for 64 keys is reserved.
<i>KEY_F(n)</i>	(<i>KEY_F0</i> +(n))
<i>KEY_DL</i>	delete line
<i>KEY_IL</i>	insert line
<i>KEY_DC</i>	delete character
<i>KEY_IC</i>	insert character or enter insert mode
<i>KEY_EC</i>	exit insert character mode
<i>KEY_CLEAR</i>	clear screen
<i>KEY_EOS</i>	clear to end of screen
<i>KEY_EOL</i>	clear to end of line
<i>KEY_SF</i>	scroll 1 line forward
<i>KEY_SR</i>	scroll 1 line backwards (reverse)
<i>KEY_NPAGE</i>	next page
<i>KEY_PPAGE</i>	previous page
<i>KEY_STAB</i>	set tab
<i>KEY_CTAB</i>	clear tab
<i>KEY_CATAB</i>	clear all tabs
<i>KEY_ENTER</i>	enter or send
<i>KEY_SRESET</i>	soft (partial) reset
<i>KEY_RESET</i>	reset or hard reset
<i>KEY_PRINT</i>	print or copy
<i>KEY_LL</i>	home down or bottom (lower left)
<i>KEY_A1</i>	upper left of virtual keypad
<i>KEY_A3</i>	upper right of virtual keypad
<i>KEY_B2</i>	center of virtual keypad
<i>KEY_C1</i>	lower left of virtual keypad
<i>KEY_C3</i>	lower right of virtual keypad

The virtual keypad is arranged like this:

A1	up	A3
left	B2	right
C1	down	C3

Functions

4.1 FUNCTION CATEGORIES

The routines can be categorised under the following headings:

- Overall Screen Manipulation
- Window and Pad Manipulation
- Output
- Input
- Output Options Setting
- Input Options Setting
- Environment Queries
- Miscellaneous

4.2 NAMING CONVENTIONS

Many of the routines have two or more versions. The routines prefixed with *w* require a *window* or *pad* argument; the exceptions are *wnoutrefresh()* and *wrefresh(win)*, which only accept a *window* argument. In these cases, *prefresh()* or *pnoutrefresh()* must be used if a *pad* is to be manipulated. The routines prefixed with *p* require a *pad* argument. Those without a prefix generally manipulate *stdscr*.

The routines prefixed with *mv* require *y* and *x* coordinates to move to, before performing the appropriate action. The *mv* routines imply a call to *move()* before the call to the other routine. The upper left corner is always (0,0), not (1,1). So, *move(y, x)*; with *y* = 1 and *x* = 0 will move the window cursor to the second line, first column.

The routines prefixed with *mvw* take both a *window* or *pad* argument and *y* and *x* coordinates. The window argument is always specified before the coordinates.

4.3 EXAMPLE OF USE

The following example illustrates the use of the *curses* package. This program displays an asterisk at a random point on the screen, waits for a space to be typed and loops. Input is read one character at a time, with echo turned off.

```
/*
** stars.c
**      curses demonstration program
**/

#include<curses.h>
#include<signal.h>

extern void      srand( );
extern void      exit( );

/*
** trap()
**      invoked on receipt of interrupt signal
**      reset terminal modes and exit
**/
trap(sig)
int      sig;
{
    (void) endwin();
    exit(sig);
}

main()
{
    int      x;
    int      y;

    /* trap interrupts */
    (void) signal(SIGINT, trap);

    /* initialise terminal */
    (void) initscr();
    (void) noecho();
    (void) cbreak();
    (void) clear();

    /* seed the random number generator */
    srand((unsigned) getpid());

    /* loop */
    for (;;) {
```

```
/* generate random coordinates */
y = rand() % LINES;
x = rand() % COLS;
/* put '*' into window */
(void) move(y, x);
(void) addch('*');
/* and display it */
(void) refresh();
/* wait for space to be typed */
while (getch() != ' ')
    ;
}
/* not reached */
}
```


Synchronous and Networked Asynchronous Terminals

These notes indicate to the application writer some considerations to be borne in mind when driving synchronous, networked asynchronous (NWA), or non-standard directly connected asynchronous terminals.

Such terminals are often used in a mainframe environment and communicate to the host in block-mode. That is, the user types characters at the terminal then presses a special key to initiate transmission of the characters to the host.

Frequently, although it may be possible to send arbitrary sized blocks to the host, it may not be possible or desirable to cause a character to be transmitted with only a single keystroke.

This can cause severe problems to an application wishing to make use of single character input; see INPUT below.

5.1 OUTPUT

The *curses* package can be used in the normal way for all operations pertaining to output to the terminal, with the possible exception that on some terminals the *refresh()* routine may have to redraw the entire screen contents in order to perform any update.

If it is additionally necessary to clear the screen before each such operation, the result could be unacceptable.

Current experience is that *curses* works in a satisfactory manner on these terminals.

5.2 INPUT

Because of the nature of operation of synchronous (block-mode) and NWA terminals, it may not be possible to support all or any of the *curses* input functions. In particular, the following points should be noted:

- Single character input may not be possible. It may be necessary to press a special key to cause all characters typed at the terminal to be transmitted to the host.
- It may not be possible to disable echo. Character echo may be performed directly by the terminal. On terminals which behave in this way, any *curses* application which performs input should be aware that any characters typed will appear on the screen at wherever the cursor is positioned. This may not necessarily correspond to the position of the cursor in the window.

Call Specifications

This chapter contains detailed descriptions of the X/OPEN *curses* functions. The following general notes apply throughout.

6.1 RETURN VALUE

All routines return the integer *OK* upon successful completion and the integer *ERR* on failure, unless otherwise noted in the descriptions.

Routines that return pointers always return the null pointer on error.

6.2 <curses.h> HEADER FILE

A number of structure types and symbolic values used by the functions described in this chapter are declared in the <curses.h> header file.

Wherever functions from this chapter are used in a program, care should be taken to ensure that <curses.h> has been included. Note that in the following definitions, it is assumed that <curses.h> has already been included.

NAME

`addch`, `waddch`, `mvaddch`, `mvwaddch` — add character to window

SYNOPSIS

```
addch(ch)
cctype ch;

waddch(win, ch)
WINDOW *win;
cctype ch;

mvaddch(y, x, ch)
int y, x;
cctype ch;

mvwaddch(win, y, x, ch)
WINDOW *win;
int y, x;
cctype ch;
```

DESCRIPTION

The character *ch* is put into the window at the current cursor position of the window and the position of the window cursor is advanced. Its function is similar to that of *putchar*. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if *scrollok()* is enabled, the scrolling region will be scrolled up one line.

If *ch* is a tab, newline, or backspace, the cursor will be moved appropriately within the window. A newline also does a *clrtoeol()* before moving. A tab moves the cursor to the next tab position within the window. If *ch* is another control character, it will be drawn in the ^X notation. Calling *winch()* after adding a control character will not return the control character, but instead will return the representation of the control character.

Video attributes can be combined with a character by *oring* them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using *inch()* and *addch()*.) See *standout()*.

APPLICATION USAGE

Addch, *mvaddch*, and *mvwaddch* are macros.

NAME

`addstr`, `waddstr`, `mvaddstr`, `mvwaddstr` — add string to window

SYNOPSIS

```
addstr(str)
char *str;

waddstr(win, str)
WINDOW *win;
char *str;

mvaddstr(y, x, str)
int y, x;
char *str;

mvwaddstr(win, y, x, str)
WINDOW *win;
int y, x;
char *str;
```

DESCRIPTION

These routines write all the characters of the null terminated character string *str* on the given window. The functionality is equivalent to calling *waddch()* once for each character in the string.

APPLICATION USAGE

Addstr, *mvaddstr*, and *mvwaddstr* are macros.

NAME

attroff, attron, attrset, standend, standout, wstandend, wstandout, wattroff, wattron, wattrset — attribute manipulation

SYNOPSIS

```
attroff(attrs)
int attrs;

wattroff(win, attrs)
WINDOW *win;
int attrs;

attron(attrs)
int attrs;

wattron(win, attrs)
WINDOW *win;
int attrs;

attrset(attrs)
int attrs;

wattrset(win, attrs)
WINDOW *win;
int attrs;

standend()

wstandend(win)
WINDOW *win;

standout()

wstandout(win)
WINDOW *win;
```

DESCRIPTION

These routines manipulate the current attributes of the named window. These attributes can be any combination of A_STANDOUT, A_REVERSE, A_BOLD, A_DIM, A_BLINK and A_UNDERLINE. These constants are defined in `< curses.h >` and can be combined with the C | (or) operator.

The current attributes of a window are applied to all characters that are written into the window with `waddch()`. Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of characters put on the screen.

Attrset(attrs) sets the current attributes of the given window to *attrs*. *Attroff(attrs)* turns off the named attributes without turning on or off any other attributes. *Attron(attrs)* turns on the named attributes without affecting any others. *standout()* is the same as *attron(A_STANDOUT)*. *Standend()* is the same as *attrset(0)*; that is, it turns off all attributes.

ATTROFF(CURSES)

User Interface

APPLICATION USAGE

Attroff, *attron*, and *attrset* are macros.

NAME

baudrate — return terminal baudrate

SYNOPSIS

baudrate()

DESCRIPTION

Returns the output speed of the terminal. The number returned is in bits per second, for example 9600, and is an integer.

NAME

beep, flash — generate audiovisual alarm

SYNOPSIS

beep()

flash()

DESCRIPTION

These routines are used to signal the terminal user. *Beep()* will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. *Flash()* will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible, nothing will happen. Nearly all terminals have an audible signal (bell or beep), but only some can flash the screen.

NAME

box — draw box

SYNOPSIS

box(win, vert, hor)

WINDOW

chtype vert, hor;

DESCRIPTION

A box is drawn around the edge of the window. *Vert* and *hor* are the characters the box is to be drawn with. If *vert* and *hor* are 0, then appropriate default characters will be used.

NAME

`cbreak` — set/clear `cbreak` mode

SYNOPSIS

`cbreak()`

`nocbreak()`

DESCRIPTION

These two routines put the terminal into and out of `CBREAK` mode. In this mode, characters typed by the user are immediately available to the program and erase/kill character processing is not performed. When out of this mode, the teletype driver will buffer characters typed until a newline or carriage return is typed. Interrupt and flow control characters are unaffected by this mode. Initially the terminal may or may not be in `CBREAK` mode, as it is inherited.

NAME

clear, wclear —clear window

SYNOPSIS

clear()

wclear(win)

WINDOW *win;

DESCRIPTION

These routines are like *erase()* and *werase()*, but they also call *clearok()*, arranging that the screen will be cleared completely on the next call to *wrefresh()* for that window and repainted from scratch.

APPLICATION USAGE

Clear is a macro.

NAME

clearok — enable screen clearing

SYNOPSIS

```
clearok(win, bf)  
WINDOW *win;  
bool bf;
```

DESCRIPTION

If *bf* is *true*, the next call to *wrefresh()* with this window will clear the screen completely and redraw the entire screen from scratch. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

NAME

clrtobot, wclrtobot — clear to end of screen

SYNOPSIS

clrtobot()

wclrtobot(win)

WINDOW *win;

DESCRIPTION

All lines below the cursor in this window are erased. The current line to the right of the cursor, inclusive, is also erased.

APPLICATION USAGE

Clrtobot is a macro.

NAME

clrtoeol, wclrtoeol — clear to end of line

SYNOPSIS

clrtoeol()

wclrtoeol(win)

WINDOW *win;

DESCRIPTION

The current line to the right of the cursor, inclusive, is erased.

APPLICATION USAGE

Clrtoeol is a macro.

NAME

def_prog_mode, def_shell_mode — save terminal modes

SYNOPSIS

def_prog_mode()

def_shell_mode()

DESCRIPTION

Save the current terminal modes as the "program" (in *curses*) or "shell" (not in *curses*) state for use by the *reset_prog_mode()* and *reset_shell_mode()* routines. This is done automatically by *initscr()*.

NAME

delay_output — cause short delay

SYNOPSIS

```
delay_output(ms)
int ms;
```

DESCRIPTION

Insert *ms* millisecond pause in output. On some systems, this function has no effect.

NAME

`delch`, `mvdelch`, `mvwdelch`, `wdelch` — remove character from window

SYNOPSIS

`delch()`

`wdelch(win)`

WINDOW *win;

`mvdelch(y, x)`

int y, x;

`mvwdelch(win, y, x)`

WINDOW *win;

int y, x;

DESCRIPTION

The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change (after moving to *y,x* if specified).

APPLICATION USAGE

Delch, *mvdelch*, and *mvwdelch* are macros.

NAME

deleteln, wdeleteln — remove line from window

SYNOPSIS

deleteln()

wdeleteln(win)

WINDOW *win;

DESCRIPTION

The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change.

APPLICATION USAGE

Deleteln is a macro.

NAME

delwin — delete window

SYNOPSIS

delwin(win)

WINDOW *win;

DESCRIPTION

Deletes the named window, freeing all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

NAME

echo, noecho — enable/disable terminal echo

SYNOPSIS

echo()

noecho()

DESCRIPTION

These routines control whether characters typed by the user are echoed by the input routine. Initially, input characters are echoed. Authors of interactive programs, using *curses*, prefer to do echoing in a controlled manner, so they disable echoing. Subsequent calls to *echo()* and *noecho()* do not flush typeahead.

NAME

endwin — restore initial terminal environment

SYNOPSIS

endwin()

DESCRIPTION

A program should always call *endwin()* before exiting or escaping from *curses* mode temporarily. This routine will restore tty modes, move the cursor to the lower left corner of the screen and reset the terminal into the proper non-visual mode. To resume after a temporary escape, *refresh()* or *doupdate()* should be called.

NAME

erase, werase — copy blanks into window

SYNOPSIS

erase()

werase(win)

WINDOW *win;

DESCRIPTION

These routines copy blanks to every position in the window.

APPLICATION USAGE

Erase is a macro.

NAME

erasechar — return current ERASE character

SYNOPSIS

erasechar()

DESCRIPTION

The user's current erase character is returned.

NAME

flushinp — discard typeahead

SYNOPSIS

flushinp()

DESCRIPTION

Throws away any typeahead that has been typed by the user and has not yet been read by the program.

NAME

getch, mvgetch, mvwgetch, wgetch — read character

SYNOPSIS

getch()

wgetch(win)

WINDOW *win;

mvgetch(y, x)

int y, x;

mvwgetch(win, y, x)

WINDOW *win;

int y, x;

DESCRIPTION

A character is read from the terminal associated with the window. In *nodelay* mode, if there is no input waiting, the value [ERR] is returned. In *delay* mode, the program will hang until the system passes text through to the program. Depending on the setting of *cbreak()*, this will be after one character or after the first newline. Unless *noecho()* has been set, the character will also be echoed into the designated window.

If *keypad()* is *true*, and a function key is pressed, the token for that function key will be returned instead of the raw characters. Possible function keys are defined in *< curses.h >* with integers beginning with 0401, whose names begin with KEY_. If a character is received that could be the beginning of a function key (such as escape), *curses* will set a timer. If the remainder of the sequence does not come in within the designated time, the character will be passed through, otherwise the function key value will be returned. For this reason, on many terminals, there will be a delay after a user presses the escape key before the escape is returned to the program. (Use by a programmer of the escape key for a single character function is discouraged.)

APPLICATION USAGE

Getch, *mvgetch*, and *mvwgetch* are macros.

NAME

getstr, mvgetstr, mvwgetstr, wgetstr — read string

SYNOPSIS

```
getstr(str)
char *str;

wgetstr(win, str)
WINDOW *win;
char *str;

mvgetstr(y, x, str)
int y, x;
char *str;

mvwgetstr(win, y, x, str)
WINDOW *win;
int y, x;
char *str;
```

DESCRIPTION

A series of characters are read until a newline or carriage return is received. The resulting value is placed in the area pointed at by the character pointer *str*. The user's erase and kill characters are interpreted.

APPLICATION USAGE

Getstr, *mvgetstr*, and *mvwgetstr* are macros.

Current implementations perform function key mapping in a manner similar to *getch*. Applications are warned, however, that this functionality is not guaranteed.

NAME

getyx — get cursor position

SYNOPSIS

```
getyx(win, y, x)
WINDOW *win;
int y, x;
```

DESCRIPTION

The cursor position of the window is placed in the two integer variables *y* and *x*. This is implemented as a macro, so no `&` is necessary before the variables.

NAME

has_ic — determine whether insert/delete character available

SYNOPSIS

has_ic()

DESCRIPTION

True if the terminal has insert- and delete-character capabilities.

APPLICATION USAGE

The functions *insch()* and *delch()* are always available in the *curses* library.

NAME

has_il — determine whether insert/delete line is available

SYNOPSIS

has_il()

DESCRIPTION

True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to check to see if it would be appropriate to turn on physical scrolling using *scrollok()*.

APPLICATION USAGE

The functions *insertln()* and *deleteln()* are always available in the *curses* library.

NAME

idlok — enable use if insert/delete line

SYNOPSIS

```
idlok(win, bf)
WINDOW *win;
bool bf;
```

DESCRIPTION

If enabled (*bf* is *true*), *curses* will use the hardware insert/delete line hardware of terminals so equipped. If disabled, *curses* will not use this feature. (The insert/delete character feature is always used.) This option should be enabled only if the application needs insert/delete line; for example, for a screen editor. It is disabled by default because insert/delete line tends to be visually annoying when used in applications where it isn't really needed. If insert/delete line cannot be used, *curses* will redraw the changed portions of all lines.

NAME

`inch`, `mvinch`, `mvwinch`, `winch` — return character from window

SYNOPSIS

```
inch(  
    winch(win)  
    WINDOW *win;  
  
    mvinch(y, x)  
    int y, x;  
  
    mvwinch(win, y, x)  
    WINDOW *win;  
    int y, x;
```

DESCRIPTION

The character of type *chtype* at the current position in the named window is returned. If any attributes are set for that position, their values will be *ored* into the value returned. The predefined constants `A_CHARTEXT` and `A_ATTRIBUTES`, defined in `< curses.h >`, can be used with the `&` (logical *and*) operator to extract the character or attributes alone.

APPLICATION USAGE

Inch, *winch*, *mvinch*, and *mvwinch* are macros.

NAME

initscr — initialise terminal environment

SYNOPSIS

initscr()

DESCRIPTION

The first routine called should almost always be *initscr()*. This will determine the terminal type and initialise all *curses* data structures. *Initscr()* also arranges that the first call to *refresh()* will clear the screen. If errors occur, *initscr()* will write an appropriate error message to standard error and exit. If the program wants an indication of error conditions, *newterm()* should be used instead of *initscr()*.

NAME

`insch`, `mvinsch`, `mvwinsch`, `winsch` — insert character

SYNOPSIS

```
insch(ch)
ctype ch;

winsch(win, ch)
WINDOW *win;
ctype ch;

mvinsch(y, x, ch)
int y, x;
ctype ch;

mvwinsch(win, y, x, ch)
WINDOW *win;
int y, x;
ctype ch;
```

DESCRIPTION

The character *ch* is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character on the line. The cursor position does not change (after moving to *y,x* if specified).

APPLICATION USAGE

Insch, *mvinsch*, and *mvwinsch* are macros.

NAME

insertln, winsertln — insert line

SYNOPSIS

insertln()

winsertln(win)

WINDOW *win;

DESCRIPTION

A blank line is inserted above the current line and the bottom line is lost.

APPLICATION USAGE

Insertln is a macro.

NAME

intrflush — enable flush on interrupt,

SYNOPSIS

```
intrflush(win, bf)
```

```
WINDOW *win;
```

```
bool bf;
```

DESCRIPTION

If this option is enabled, when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt but causing *curses* to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

NAME

keypad — enable keypad

SYNOPSIS

```
keypad(win, bf)
WINDOW *win;
bool bf;
```

DESCRIPTION

This option enables the keypad of the user's terminal. If enabled, the user can press a function key (such as an arrow key) and *getch()* will return a single value representing the function key, as in `KEY_LEFT`. (See Function Keys.) If disabled, *curses* will not treat function keys specially and the program would have to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option will cause the terminal keypad to be turned on before input is done.

APPLICATION USAGE

The keypad layout is very terminal dependent; some terminals might not even have a keypad.

NAME

killchar — return current kill character

SYNOPSIS

killchar()

DESCRIPTION

The user's current line kill character is returned.

NAME

leaveok — enable non-tracking cursor

SYNOPSIS

```
leaveok(win, bf)  
WINDOW *win;  
bool bf;
```

DESCRIPTION

Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

NAME

longname — return full terminal type name

SYNOPSIS

char *longname()

DESCRIPTION

This routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to *initscr()* or *newterm()*. The area is overwritten by each call to *newterm()* and is not restored by *set_term()*, so the value should be saved between calls to *newterm()* if *longname()* is going to be used with multiple terminals.

NAME

move, wmove — move cursor in window

SYNOPSIS

move(y, x)

wmove(win, y, x)

WINDOW *win;

int y, x;

DESCRIPTION

The cursor associated with the window is moved to the given location. This does not move the physical cursor of the terminal until *refresh()* is called. The position specified is relative to the upper left corner of the window, which is (0,0).

APPLICATION USAGE

Move is a macro.

NAME

mvwin — move window

SYNOPSIS

```
mvwin(win, y, x)
WINDOW *win;
int y, x;
```

DESCRIPTION

Move the window so that the upper left corner will be at position (y, x). If the move would cause the window to be off the screen, it is an error and the window is not moved.

NAME

newpad — create new pad

SYNOPSIS

WINDOW *newpad(nlines, ncols)
int nlines, ncols;

DESCRIPTION

Creates a new *pad* data structure. A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g. from scrolling or echoing of input) do not occur. It is not legal to call *refresh()* with a pad as an argument; the routines *prefresh()* or *pnoutrefresh()* should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

NAME

newterm — open new terminal

SYNOPSIS

SCREEN *newterm(type, outfd, infd)

char *type;

FILE *outfd, *infd;

DESCRIPTION

A program which outputs to more than one terminal should use *newterm()* for each terminal instead of *initscr()*. A program which wants an indication of error conditions, so that it may continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, would also use this routine. The routine *newterm()* should be called once for each terminal. It returns a variable of type SCREEN * which should be saved as a reference to that terminal. The arguments are the type of the terminal to be used in place of TERM, a file pointer for output to the terminal, and another file pointer for input from the terminal. The program must also call *endwin()* for each terminal being used when it is done running.

NAME

newwin — create new window

SYNOPSIS

```
WINDOW *newwin(nlines, ncols, begin_y, begin_x)
int nlines, ncols, begin_y, begin_x;
```

DESCRIPTION

Create a new window with the given number of lines, *nlines*, and columns, *ncols*. The upper left corner of the window is at line *begin_y*, column *begin_x*. If either *nlines* or *ncols* is zero, they will be defaulted to LINES - *begin_y* and COLS - *begin_x*. A new full-screen window is created by calling *newwin*(0,0,0,0).

NAME

nl, nonl — enable/disable newline control

SYNOPSIS

nl()

nonl()

DESCRIPTION

These routines control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations, *curses* is able to make better use of the linefeed capability, resulting in faster cursor motion.

APPLICATION USAGE

Nl is a macro.

NAME

nodelay — disable block during read

SYNOPSIS

nodelay(win, bf)

WINDOW *win;

bool bf;

DESCRIPTION

This option causes *getch()* to be a non-blocking call. If no input is ready, *getch()* will return ERR. If disabled, *getch()* will hang until input is ready.

NAME

overlay, overwrite — overlay windows

SYNOPSIS

overlay(srcwin, dstwin)

WINDOW *srcwin, *dstwin;

overwrite(srcwin, dstwin)

WINDOW *srcwin, *dstwin;

DESCRIPTION

These routines overlay *srcwin* on top of *dstwin*; that is, all text in *srcwin* is copied into *dstwin*. *Srcwin* and *dstwin* are not required to be the same size. The copy starts at (0,0) on each window. The difference is that *overlay()* is non-destructive (blanks are not copied) while *overwrite()* is destructive.

NAME

prefresh, pnoutrefresh — refresh pad

SYNOPSIS

```
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
```

```
WINDOW *pad;
```

```
int pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol;
```

```
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
```

```
WINDOW *pad;
```

```
int pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol;
```

DESCRIPTION

These routines are analogous to *wrefresh()* and *wnoutrefresh()* except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. *Pminrow* and *pmincol* specify the upper left corner, in the pad, of the rectangle to be displayed. *Sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures.

NAME

`printw`, `mvprintw`, `mvwprintw`, `wprintw` — formatted write to a window

SYNOPSIS

```
printw(fmt [, arg] ...)
char *fmt;
```

```
wprintw(win, fmt [, arg] ...)
WINDOW *win;
char *fmt;
```

```
mvprintw(y, x, fmt [, arg] ...)
int y, x;
char *fmt;
```

```
mvwprintw(win, y, x, fmt [, arg] ...)
WINDOW *win;
int y, x;
char *fmt;
```

DESCRIPTION

These routines are analogous to *printf*. The string which would be output by *printf()* is instead output using *waddstr()* on the given window.

NAME

`reset_prog_mode`, `reset_shell_mode` — restore terminal mode

SYNOPSIS

`reset_prog_mode()`

`reset_shell_mode()`

DESCRIPTION

Restore the terminal to "program" (in *curses*) or "shell" (out of *curses*) state. These are done automatically by `endwin()` and `doupdate()` after an `endwin()`, so they would normally not be called before.

NAME

raw, noraw — enable/disable raw mode

SYNOPSIS

raw()

noraw()

DESCRIPTION

The terminal is placed into or out of raw mode. Raw mode is similar to CBREAK mode, in that characters typed are immediately passed through to the user program. The differences are that in RAW mode, the interrupt, quit, suspend and flow control characters are passed through uninterpreted, instead of generating a signal. The behaviour of the BREAK key depends on other bits that are not set by *curses*.

NAME

refresh, wrefresh — refresh window

SYNOPSIS

refresh()

wrefresh(win)

WINDOW *win;

DESCRIPTION

These routines must be called to get any output on the terminal, as other routines merely manipulate data structures. *Wrefresh()* copies the named window to the physical terminal screen, taking into account what is already there in order to do optimisations. *Refresh()* is the same, using *stdscr* as a default screen. Unless *leaveok()* has been enabled, the physical cursor of the terminal is left at the location of the window's cursor.

APPLICATION USAGE

Refresh is a macro.

NAME

resetty, savetty — save/restore terminal modes

SYNOPSIS

resetty()

savetty()

DESCRIPTION

These routines save and restore the state of the terminal modes. *Savetty()* saves the current state in a buffer and *resetty()* restores the state to what it was at the last call to *savetty()*.

NAME

scanw, mvscanw, mvwscanw, wscanw — formatted read from window

SYNOPSIS

```
scanw(fmt [, arg] ...)
```

```
char *fmt;
```

```
wscanw(win, fmt [, arg] ...)
```

```
WINDOW *win;
```

```
char *fmt;
```

```
mvscanw(y, x, fmt [, arg] ...)
```

```
int y, x;
```

```
char *fmt;
```

```
mvwscanw(win, y, x, fmt [, arg] ...)
```

```
WINDOW *win;
```

```
int y, x;
```

```
char *fmt;
```

DESCRIPTION

These routines correspond to *scanf*. *Wgetstr()* is called on the window, and the resulting line is used as input for the scan.

NAME

scroll — scroll window

SYNOPSIS

scroll(win)

WINDOW *win;

DESCRIPTION

The window is scrolled up one line. This involves moving the lines in the window data structure.

NAME

scrollok — enable screen scrolling

SYNOPSIS

scrollok(win, bf)

WINDOW *win;

bool bf;

DESCRIPTION

This option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either from a newline on the bottom line, or typing the last character of the last line. If disabled, the cursor is left on the bottom line. If enabled, the window is scrolled up one line and then refreshed.

NAME

set_term — switch between terminals

SYNOPSIS

```
SCREEN *set_term(new)
SCREEN *new;
```

DESCRIPTION

This routine is used to switch between different terminals. The screen reference *new* becomes the new current terminal. The previous terminal is returned by the routine. This is the only routine which manipulates SCREEN pointers; all other routines affect only the current terminal.

NAME

setscrreg, wsetscrreg — set scrolling region

SYNOPSIS

setscrreg(top, bot)

int top, bot;

wsetscrreg(win, top, bot)

WINDOW *win;

int top, bot;

DESCRIPTION

These routines allow the user to set a software scrolling region in a window. *Top* and *bot* are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and *scrollok()* are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. Only the text of the window is scrolled.

NAME

subwin — create subwindow

SYNOPSIS

```
WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
```

```
WINDOW *orig;
```

```
int nlines, ncols, begin_y, begin_x;
```

DESCRIPTION

Create a new window with the given number of lines, *nlines*, and columns, *ncols*. The window is at position (*begin_y*, *begin_x*) on the screen. (This position is relative to the screen, and not to the window *orig*.) The window is made in the middle of the window *orig*, so that changes made to one window will affect both windows. When using this routine it will often be necessary to call *touchwin()* before calling *wrefresh()*.

NAME

touchwin — touch window

SYNOPSIS

```
touchwin(win)
WINDOW *win;
```

DESCRIPTION

Throw away all optimisation information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change.

NAME

typeahead — check for typeahead

SYNOPSIS

```
typeahead(fd)
int fd;
```

DESCRIPTION

Curses does "line-breakout optimisation" by looking for typeahead periodically while updating the screen. If input is found, the current update will be postponed until *refresh()* or *doupdate()* is called again. This allows faster response to commands typed in advance. Normally, the input FILE pointer passed to *newterm()*, or *stdin* in the case that *initscr()* was used, will be used to do this typeahead checking. The *typeahead()* routine specifies that the file descriptor *fd* is to be used to check for typeahead instead. If *fd* is -1, then no typeahead checking will be done.

NAME

unctrl — convert character to printable form

SYNOPSIS

```
unctrl(c)
chtype c;
```

DESCRIPTION

This macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the ^X notation. Printing characters are displayed as is.

APPLICATION USAGE

Unctrl is a macro, which is defined in `<unctrl.h>`.

NAME

douupdate, wnoutrefresh — do efficient refresh

SYNOPSIS

wnoutrefresh(win)

WINDOW *win;

douupdate()

DESCRIPTION

These two routines allow multiple updates with more efficiency than *wrefresh()* alone. In addition to all of the window structures, *curses* keeps two data structures representing the terminal screen: a physical screen, describing what is actually on the screen, and a virtual screen, describing what the programmer wants to have on the screen.

The routine *wrefresh()* works by first calling *wnoutrefresh()*, which copies the named window to the virtual screen, and then calling *douupdate()*, which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to *wrefresh()* will result in alternating calls to *wnoutrefresh()* and *douupdate()*, causing several bursts of output to the screen. By first calling *wnoutrefresh()* for each window, it is then possible to call *douupdate()* once, resulting in only one burst of output, with probably fewer total characters transmitted and certainly less CPU time used.

X/O/P/E/N/

PORTABILITY GUIDE

THE X/OPEN SYSTEM V SPECIFICATION
INTER-PROCESS COMMUNICATION

Contents

Chapter 1 INTER-PROCESS COMMUNICATION

1.1 INTRODUCTION

1.2 EFFECTS ON "XVS SYSTEM CALLS AND LIBRARIES"

Chapter 2 IPC SERVICES

<i>ipc(2)</i>	OPTIONAL
<i>msgctl(2)</i>	OPTIONAL
<i>msgget(2)</i>	OPTIONAL
<i>msgop(2)</i>	OPTIONAL
<i>semctl(2)</i>	OPTIONAL
<i>semget(2)</i>	OPTIONAL
<i>semop(2)</i>	OPTIONAL
<i>shmctl(2)</i>	OPTIONAL
<i>shmget(2)</i>	OPTIONAL
<i>shmop(2)</i>	OPTIONAL
<i>ipc(5)</i>	
<i>msg(5)</i>	
<i>sem(5)</i>	
<i>shm(5)</i>	

Inter-Process Communication

1.1 INTRODUCTION

This Part of the Portability Guide describes the Inter-Process Communication (IPC) services available in X/OPEN Systems.

The routines in System V which provide IPC are used by a number of existing applications programs. X/OPEN are considering the problem of generalised IPC, which is not fully addressed by the System V IPC. For example, the shared memory functionality is hard to implement efficiently on multiprocessor architectures or across networks.

The System V IPC has been included in this issue of the XVS, but should not be considered to be a firm commitment to support this functionality in the future. Applications Developers who need to use IPC are warned not to base architectural decisions on this particular form of IPC, and to design their applications so that modules using these routines can easily be modified to use alternative methods at a later date.

1.2 EFFECTS ON "XVS SYSTEM CALLS AND LIBRARIES"

The adoption of IPC affects some of the services described in "XVS SYSTEM CALLS AND LIBRARIES". The services affected are shown in the table below.

Services affected by IPC	
<i>exec(2)</i>	<i>exit(2)</i>
<i>fork(2)</i>	<i>errno(2)</i>



Chapter 2

IPC Services

This chapter describes the IPC services. They are described in the same format as the entries in "XVS SYSTEM CALLS AND LIBRARIES".

NAME

ipc — description of ipc (OPTIONAL)

DESCRIPTION

The message-passing, semaphore and shared memory services form between them the Inter-Process Communication (IPC) facility. Certain aspects of their operation are common, and are described below.

Each individual shared memory segment, message queue and semaphore set is identified by a unique positive integer, called respectively a shared memory identifier *shmid*, a semaphore identifier, *semid*, and a message queue identifier, *msqid*. The identifiers are returned by calls on *shmget*(2), *semget*(2) and *msgget*(2), respectively.

Associated with each identifier is a data structure which contains data related to the operations which may be or have been performed. See *shm*(5), *sem*(5) and *msg*(5) for their descriptions.

Each of the data structures contains both ownership information and an *ipc_perm* structure, see *ipc*(5), which are used in conjunction to determine whether or not read/write (read/alter for semaphores) permissions should be granted to processes using the IPC facilities. The *mode* member of the *ipc_perm* structure acts as a bit field which determines the permissions. The values of the bits are given below.

Bit	Meaning
0400	Read by user
0200	Write by user
0040	Read by group
0020	Write by group
0004	Read by others
0002	Write by others

The name of the *ipc_perm* structure is *shm_perm*, *sem_perm*, or *msg_perm* depending on which service is being used. In each case, read and write/alter permissions are granted to a process if one or more of the following are true (*xxx* is replaced by *shm*, *sem*, or *msg* appropriately):

- The effective user ID of the process is super-user.
- The effective user ID of the process matches *xxx_perm.cuid* or *xxx_perm.uid* in the data structure associated with the IPC identifier and the appropriate bit of the *user* field in *xxx_perm.mode* is set.
- The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* but the effective group ID of the process matches *xxx_perm.cgid* or *xxx_perm.gid* in the data structure associated with the IPC identifier, and the appropriate bit of the *group* field in *xxx_perm.mode* is set.

- The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* and the effective group ID of the process does not match *xxx_perm.cgid* or *xxx_perm.gid* in the data structure associated with the IPC identifier, but the appropriate bit of the *other* field in *xxx_perm.mode* is set.

Otherwise, the permission is denied.

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Not in the SVID. This descriptive entry is specific to the X/OPEN Portability Guide.

NAME

msgctl — message-control-operations (OPTIONAL)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

DESCRIPTION

The function *msgctl* provides a variety of message-control-operations as specified by *cmd*. The following values for *cmd* and the message-control-operations they specify are available:

IPC_STAT

Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in *msg(5)*.

IPC_SET

Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9-bits */
msg_qbytes
```

This *cmd* can only be executed by a process that has an effective-user-ID equal to either that of super-user or to the value of *msg_perm.cuid* or *msg_perm.uid* in the data structure associated with *msqid*. Only super-user can raise the value of *msg_qbytes*.

IPC_RMID

Remove the message-queue-identifier specified by *msqid* from the system and destroy the message-queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective-user-ID equal to either that of super-user or to the value of *msg_perm.cuid* or *msg_perm.uid* in the data structure associated with *msqid*.

RETURN VALUE

If successful, the function *msgctl* returns 0; otherwise, it returns -1 and *errno* will indicate the error.

ERRORS

The function *msgctl* will fail if one or more of the following are true:

[EINVAL]

The value of *msgid* is not a valid message-queue-identifier; or the value of *cmd* is not a valid command.

[EACCES]

The argument *cmd* is equal to `IPC_STAT` and the calling-process does not have read permission, see *ipc(2)*.

[EPERM]

The argument *cmd* is equal to `IPC_RMID` or `IPC_SET` and the effective-user-ID of the calling-process is not equal to that of super-user and it is not equal to the value of *msg_perm.cuid* or *msg_perm.uid* in the data structure associated with *msgid*.

[EPERM]

The argument *cmd* is equal to `IPC_SET`, an attempt is being made to increase to the value of *msg_qbytes*, and the effective-user-ID of the calling-process is not equal to that of super-user.

SEE ALSO

ipc(2), *msgget(2)*, *msgop(2)*, *msg(5)*.

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from the entry in Issue 2 of the SVID.

NAME

`msgget` — get message-queue (OPTIONAL)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key, msgflg)
key_t key;
int msgflg;
```

DESCRIPTION

The function `msgget` returns the message-queue-identifier associated with the argument `key`.

A message-queue-identifier and associated message-queue and data structure, see `msg(5)`, are created for the argument `key` if one of the following is true:

- if the argument `key` is equal to `IPC_PRIVATE`;

- if the argument `key` does not already have a message-queue-identifier associated with it, and `(msgflg & IPC_CREAT)` is non-zero.

Upon creation, the data structure associated with the new message-queue-identifier is initialised as follows:

- `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid` and `msg_perm.gid` are set equal to the effective-user-ID and effective-group-ID, respectively, of the calling-process;

- the low-order 9-bits of `msg_perm.mode` are set equal to the low-order 9-bits of `msgflg`;

- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are set equal to 0;

- `msg_ctime` is set equal to the current-time;

- `msg_qbytes` is set equal to the system-limit.

RETURN VALUE

If successful, the function `msgget` returns a non-negative integer, namely a message-queue-identifier; otherwise, it returns -1 and `errno` will indicate the error.

ERRORS

The function `msgget` will fail if one or more of the following are true:

[EACCES]

- A message-queue-identifier exists for the argument `key`, but operation permission as specified by the low-order 9-bits of `msgflg` would not be granted, see `ipc(2)`.

MSGGET(2)

System Calls

[ENOENT]

A message-queue-identifier does not exist for the argument *key* and (*msgflg* & *IPC_CREAT*) is equal to zero.

[ENOSPC]

A message-queue-identifier is to be created but the system-imposed limit on the maximum number of allowed message-queue-identifiers system-wide would be exceeded.

[EEXIST]

A message-queue-identifier exists for the argument *key* but $((msgflg \& IPC_CREAT) \&\& (msgflg \& IPC_EXCL))$ is non-zero.

SEE ALSO

ipc(2), msgctl(2), msgop(2), msg(5).

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from the entry in Issue 2 of the SVID.

NAME

`msgsnd`, `msgrcv` — message operations (OPTIONAL)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(msqid, msgp, msgsz, msgflg)
int msqid;
struct mymsg *msgp;
int msgsz, msgflg;

int msgrcv(msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct mymsg *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

DESCRIPTION

`msgsnd`

The function `msgsnd` is used to send a message to the queue associated with the message queue identifier specified by `msqid`.

The argument `msgp` points to a user-defined buffer that must contain first a field of type `long` that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer might look like.

```
struct mymsg {
    long mtype; /* message type */
    char mtext[]; /* message text */
}
```

The structure member `mtype` is a non-zero positive integer that can be used by the receiving process for message selection (see `msgrcv` below).

The structure member `mtext` is any text of length `msgsz` bytes. The argument `msgsz` can range from zero to a system-imposed maximum.

The argument `msgflg` specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to `msg_qbytes`, see `msg(5)`.

- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

If (*msgflg* & *IPC_NOWAIT*) is non-zero, the message will not be sent and the calling-process will return immediately.

If (*msgflg* & *IPC_NOWAIT*) is equal to zero, the calling-process will suspend execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

The message-queue-identifier *msqid* is removed from the system, see *msgctl*(2). When this occurs, *errno* is set equal to [EIDRM] and a value of -1 is returned.

The calling-process receives a signal that is to be caught. In this case the message is not sent and the calling-process resumes execution in the manner prescribed in the *signal*(2) routine.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*, see *msg*(5).

msg_qnum is incremented by 1 .

msg_lspid is set equal to the process ID of the calling-process.

msg_stime is set equal to the current time.

msgrcv

The function *msgrcv* reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the user-defined buffer pointed to by *msgp*. The buffer must contain a message type field followed by the area for the message text (see the structure *mymsg* above).

The structure member *mtype* is the received message's type as specified by the sending process.

The structure member *mtext* is the text of the message.

The argument *msgsz* specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & *MSG_NOERROR*) is non-zero. The truncated part of the message is lost and no indication of the truncation is given to the calling-process.

The argument *msgtyp* specifies the type of message requested as follows:

If *msgtyp* is equal to zero, the first message on the queue is received.

If *msgtyp* is greater than zero, the first message of type *msgtyp* is received.

If *msgtyp* is less than zero, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

The argument *msgflg* specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

If (*msgflg* & *IPC_NOWAIT*) is non-zero, the calling-process will return immediately with a return value of -1 and *errno* set to [ENOMSG].

If (*msgflg* & *IPC_NOWAIT*) is equal to zero, the calling-process will suspend execution until one of the following occurs:

A message of the desired type is placed on the queue.

The message queue identifier *msqid* is removed from the system. When this occurs, *errno* is set equal to [EIDRM] and a value of -1 is returned.

The calling-process receives a signal that is to be caught. In this case a message is not received and the calling-process resumes execution in the manner prescribed in *signal(2)*.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*.

msg_qnum is decremented by 1.

msg_lripid is set equal to the process ID of the calling-process.

msg_rtime is set equal to the current time.

RETURN VALUE

If successful, the function *msgsnd* returns a value of zero.

If successful, the function *msgrcv* returns a value equal to the number of bytes actually placed into the buffer *mtext*.

Otherwise, the function *msgsnd* and the function *msgrcv* return -1 and *errno* will indicate the error.

ERRORS

The function *msgsnd* will fail and no message will be sent if one or more of the following are true:

[EINVAL]

The value of *msqid* is not a valid message-queue-identifier, or the value of *mtype* is less than 1; or the value of *msgsz* is less than zero or greater than the system-imposed limit.

[EACCES]

Operation permission is denied to the calling-process.

[EAGAIN]

The message cannot be sent for one of the reasons cited above and (*msgflg* & *IPC_NOWAIT*) is non-zero.

[EINTR]

The function *msgsnd* was interrupted by a signal.

[EIDRM]

The message-queue-identifier *msgid* has been removed from the system.

The function *msgrcv* will fail and no message will be received if one or more of the following are true:

[EINVAL]

The value of *msgid* is not a valid message-queue-identifier; or the value of *msgsz* is less than zero.

[EACCES]

Operation permission is denied to the calling-process.

[EINTR]

The function *msgrcv* was interrupted by a signal.

[EIDRM]

The message-queue-identifier *msgid* has been removed from the system.

[E2BIG]

The value of *mtext* is greater than *msgsz* and (*msgflg* & *MSG_NOERROR*) is equal to zero.

[ENOMSG]

The queue does not contain a message of the desired type and (*msgtyp* & *IPC_NOWAIT*) is non-zero.

SEE ALSO

msgctl(2), *msgget*(2), *signal*(2), *msg*(5).

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from the entry in Issue 2 of the SVID.

NAME

semctl — semaphore-control-operations (OPTIONAL)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
} arg;
```

DESCRIPTION

The function *semctl* provides a variety of semaphore-control-operations as specified by *cmd*.

The following semaphore-control-operations as specified by *cmd* are executed with respect to the semaphore specified by *semid* and *semnum*. The level of permission required for each operation is shown with each command, see *ipc(2)*. The symbolic names for the values of *cmd* are defined by the *<sys/sem.h>* header file.

GETVAL

Return the value of *semval*, see *sem(5)*.
Requires read permission.

SETVAL

Set the value of *semval* to *arg.val*. When this *cmd* is successfully executed, the *semadj* value corresponding to the specified semaphore in all processes is cleared.
Requires alter permission, see *ipc(2)*.

GETPID

Return the value of *sempid*.
Requires read permission.

GETNCNT

Return the value of *semncnt*.
Requires read permission.

GETZCNT

Return the value of *semzcnt*.
Requires read permission.

The following *cmds* operate on each *semval* in the set of semaphores.

GETALL

Return *semvals* and place into the array pointed to by *arg.array*.
Requires read permission.

SETALL

Set *semvals* according to the array pointed to by *arg.array*. When this *cmd* is successfully executed, the *semadj* values corresponding to each specified semaphore in all processes are cleared.

Requires alter permission.

The following *cmds* are also available:

IPC_STAT

Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. The contents of this structure are defined in *sem(5)*.

Requires read permission.

IPC_SET

Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode /* only low 9-bits */
```

This *cmd* can only be executed by a process that has an effective-user-ID equal to either that of super-user or to the value of *sem_perm.cuid* or *sem_perm.uid* in the data structure associated with *semid*.

IPC_RMID

Remove the semaphore-identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This *cmd* can only be executed by a process that has an effective-user-ID equal to either that of super-user or to the value of *sem_perm.cuid* or *sem_perm.uid* in the data structure associated with *semid*.

RETURN VALUE

If successful, the value *semctl* returns depends on *cmd* as follows:

GETVAL the value of *semval*.

GETPID the value of *sempid*.

GETNCNT the value of *semncnt*.

GETZCNT the value of *semzcnt*.

All others a value of zero.

Otherwise, *semctl* returns -1 and *errno* indicates the error.

ERRORS

The function *semctl* will fail if one or more of the following are true:

[EINVAL]

The value of *semid* is not a valid semaphore-identifier, or the value of *semnum* is less than zero or greater than *sem_nsems*, or the value of *cmd* is not a valid command.

[EACCES]

Operation permission is denied to the calling-process, see *ipc(2)*.

[ERANGE]

The argument *cmd* is equal to SETVAL or SETALL and the value to which *semval* is to be set is greater than the system imposed maximum.

[EPERM]

The argument *cmd* is equal to IPC_RMID or IPC_SET and the effective-user-ID of the calling-process is not equal to that of super-user and it is not equal to the value of *sem_perm.cuid* or *sem_perm.uid* in the data structure associated with *semid*.

SEE ALSO

ipc(2), *semget(2)*, *semop(2)*, *sem(5)*.

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from the entry in Issue 2 of the SVID.

NAME

`semget` — get set of semaphores (OPTIONAL)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key, nsems, semflg)
key_t key;
int nsems, semflg;
```

DESCRIPTION

The function *semget* returns the semaphore-identifier associated with *key*.

A semaphore-identifier with its associated *semid_ds* data structure and its associated set of *nsems* semaphores, see *sem(5)*, are created for *key* if one of the following are true:

The argument *key* is equal to `IPC_PRIVATE`.

The argument *key* does not already have a semaphore-identifier associated with it and (*semflg* & `IPC_CREAT`) is non-zero.

Upon creation, the *semid_ds* data structure associated with the new semaphore-identifier is initialised as follows:

In the operation permissions structure *sem_perm.cuid*, *sem_perm.uid*, *sem_perm.cgid*, and *sem_perm.gid* are set equal to the effective-user-ID and effective-group-ID, respectively, of the calling-process.

The low-order 9-bits of *sem_perm.mode* are set equal to the low-order 9-bits of *semflg*.

The variable *sem_nsems* is set equal to the value of *nsems*.

The variable *sem_otime* is set equal to 0 and *sem_ctime* is set equal to the current time.

The data structure associated with each semaphore in the set is not initialised. The function *semctl* with the command `SETVAL` or `SETALL` can be used to initialise each semaphore.

RETURN VALUE

If successful, the function *semget* returns a non-negative integer, namely a semaphore-identifier; otherwise, it returns -1 and *errno* will indicate the error.

ERRORS

The function *semget* will fail if one or more of the following are true:

[EACCES]

A semaphore-identifier exists for *key*, but operation permission as specified by the low-order 9-bits of *semflg* would not be granted.

[EINVAL]

The value of *nsems* is either less than or equal to 0 or greater than the system-imposed limit, or a semaphore-identifier exists for the argument *key*, but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to 0.

[ENOENT]

A semaphore-identifier does not exist for the argument *key* and (*semflg* & *IPC_CREAT*) is equal to zero (0).

[ENOSPC]

A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system-wide would be exceeded.

[EEXIST]

A semaphore-identifier exists for the argument *key* but ((*semflg* & *IPC_CREAT*) && (*semflg* & *IPC_EXCL*)) is non-zero.

SEE ALSO

semctl(2), *semop*(2), *sem*(5).

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from the entry in Issue 2 of the SVID.

NAME

semop — semaphore operations (OPTIONAL)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(semid, sops, nsops)
int semid;
struct sembuf *sops;
unsigned nsops;
```

DESCRIPTION

The function *semop* is used to automatically perform an user-defined array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by the argument *semid*.

The argument *sops* is a pointer to a user-defined array of semaphore-operation structures.

The argument *nsops* is the number of such structures in the array.

Each structure, *sembuf*, includes the following members:

```
short sem_num;    /* semaphore number */
short sem_op;     /* semaphore operation */
short sem_flg;    /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

The variable *sem_op* specifies one of three semaphore operations:

1. If *sem_op* is a negative integer and the calling process has alter permission, one of the following will occur:
 - If *semval*, see *sem(5)*, is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg* & *SEM_UNDO*) is non-zero, the absolute value of *sem_op* is added to the calling-process' *semadj* value for the specified semaphore, see *exit(2)*.
 - If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & *IPC_CREAT*) is non-zero, *semop* will return immediately.

- If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & *IPC_CREAT*) is equal to zero, *semop* will increment the *semncnt* associated with the specified semaphore and suspend execution of the calling-process until one of the following conditions occur:
 - The value of *semval* becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from *semval* and, if (*sem_flg* & *SEM_UNDO*) is non-zero, the absolute value of *sem_op* is added to the calling-process' *semadj* value for the specified semaphore.
 - The *semid* for which the calling-process is awaiting action is removed from the system, see *semctl*(2). When this occurs, *errno* is set equal to [EIDRM] and a value of -1 is returned.
 - The calling-process receives a signal that is to be caught. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, and the calling-process resumes execution in the manner prescribed in the routines defined in *signal*(2).
- 2. If *sem_op* is a positive integer and the calling process has alter permission, the value of *sem_op* is added to *semval* and, if (*sem_flg* & *SEM_UNDO*) is non-zero, the value of *sem_op* is subtracted from the calling-process' *semadj* value for the specified semaphore.
- 3. If *sem_op* is zero and the calling process has read permission, one of the following will occur:
 - If *semval* is zero, the function *semop* will return immediately.
 - If *semval* is not equal to zero and (*sem_flg* & *IPC_CREAT*) is non-zero, the function *semop* will return immediately.
 - If *semval* is not equal to zero and (*sem_flg* & *IPC_CREAT*) is equal to zero, the function *semop* will increment the *semzcnt* associated with the specified semaphore and suspend execution of the calling-process until one of the following occurs:
 - The value of *semval* becomes zero, at which time the value of *semzcnt* associated with the specified semaphore is decremented.
 - The *semid* for which the calling-process is awaiting action is removed from the system. When this occurs, *errno* is set equal to [EIDRM] and a value of -1 is returned.
 - The calling-process receives a signal that is to be caught. When this occurs, the value of *semzcnt* associated with the specified semaphore is decremented, and the calling-process resumes execution in the manner prescribed in the routines defined in *signal*(2).

Upon successful completion, the value of *sempid* for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling-process.

RETURN VALUE

If successful, the function *semop* returns 0; otherwise, it returns -1 and *errno* will indicate the error.

ERRORS

The function *semop* will fail if one or more of the following are true for any of the semaphore operations specified by *sops*:

[EINVAL]

The value of *semid* is not a valid semaphore-identifier, or the number of individual semaphores for which the calling-process requests a SEM_UNDO would exceed the limit.

[EFBIG]

The value of *sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.

[E2BIG]

The value of *nsops* is greater than the system-imposed maximum.

[EACCES]

Operation permission is denied to the calling-process, see *ipc(2)*.

[EAGAIN]

The operation would result in suspension of the calling-process but (*sem_flg* & *IPC_CREAT*) is non-zero.

[ENOSPC]

The limit on the number of individual processes requesting a SEM_UNDO would be exceeded.

[ERANGE]

An operation would cause a *semval* to overflow the system-imposed limit, or an operation would cause a *semadj* value to overflow the system-imposed limit.

[EINTR]

The function *semop* was interrupted by a signal.

[EIDRM]

The semaphore identifier *semid* has been removed from the system.

SEE ALSO

exec(2), *exit(2)*, *fork(2)*, *ipc(2)*, *semctl(2)*, *semget(2)*, *sem(5)*.

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from the entry in Issue 2 of the SVID.

NAME

shmctl — shared-memory-control-operations (OPTIONAL)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(shmid, cmd, buf)
int shmid, cmd;
struct shmids *buf;
```

DESCRIPTION

The function *shmctl* provides a variety of shared-memory-control-operations as specified by *cmd*. The following values for *cmd* are available:

IPC_STAT

Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*.

IPC_SET

Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9-bits */
```

This *cmd* can only be executed by a process that has an effective-user-ID equal to either that of super-user or to the value of *shm_perm.cuid* or *shm_perm.uid* in the data structure associated with *shmid*.

IPC_RMID

Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective-user-ID equal to either that of super-user or to the value of *shm_perm.cuid* or *shm_perm.uid* in the data structure associated with *shmid*.

RETURN VALUE

If successful, the function *shmctl* returns 0; otherwise, it returns -1 and *errno* will indicate the error.

ERRORS

The function *shmctl* will fail if one or more of the following are true:

[EINVAL]

The value of *shmid* is not a valid shared-memory-identifier, or the value of *cmd* is not a valid command.

[EACCES]

The argument *cmd* is equal to IPC_STAT and the calling-process does not have read permission, see *ipc(2)*.

[EPERM]

The argument *cmd* is equal to `IPC_RMID` or `IPC_SET` and the effective-user-ID of the calling-process is not equal to that of super user and it is not equal to the value of *shm_perm.cuid* or *shm_perm.uid* in the data structure associated with *shmid*.

APPLICATION USAGE

The functions *shmctl*, *shmget*, and *shmat* and *shmdt* are hardware-dependent and may not be present on all systems. The shared memory routines should not be used by applications except when extreme performance considerations require them.

SEE ALSO

`ipc(2)`, `shmget(2)`, `shmop(2)`, `shm(5)`.

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from the entry in Issue 2 of the SVID.

NAME

shmget — get shared-memory-segment (OPTIONAL)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key, size, shmflg)
key_t key;
int size, shmflg;
```

DESCRIPTION

The function *shmget* returns the shared memory identifier associated with *key*.

A shared-memory-identifier and associated data structure and shared-memory-segment of at least *size* bytes, see *shm(5)*, are created for *key* if one of the following are true:

The argument *key* is equal to *IPC_PRIVATE*;

The argument *key* does not already have a shared-memory-identifier associated with it and (*shmflg* & *IPC_CREAT*) is non-zero.

Upon creation, the data structure associated with the new shared-memory-identifier is initialised as follows:

The value of *shm_perm.cuid*, *shm_perm.uid*, *shm_perm.cgid* and *shm_perm.gid* are set equal to the effective-user-ID and effective-group-ID, respectively, of the calling-process.

The low-order 9-bits of *shm_perm.mode* are set equal to the low-order 9-bits of *shmflg*. The argument *shm_segsz* is set equal to the value of *size*.

The value of *shm_lpid*, *shm_nattch*, *shm_atime* and *shm_dtime* are set equal to zero.

The value of *shm_ctime* is set equal to the current time.

RETURN VALUE

If successful, the function *shmget* returns a non-negative integer, namely a shared-memory-identifier; otherwise, it returns -1 and *errno* will indicate the error.

ERRORS

The function *shmget* will fail if one or more of the following are true:

[EINVAL]

The value of *size* is less than the system-imposed minimum or greater than the system-imposed maximum, or a shared-memory-identifier exists for the argument *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero.

[EACCES]

A shared-memory-identifier exists for *key* but operation permission as specified by the low-order 9-bits of *shmflg* would not be granted.

[ENOENT]

A shared-memory-identifier does not exist for the argument *key* and (*shmflg* & *IPC_CREAT*) is equal to zero.

[ENOSPC]

A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system-wide would be exceeded.

[ENOMEM]

A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request.

[EEXIST]

A shared-memory-identifier exists for the argument *key* but ((*shmflg* & *IPC_CREAT*) && (*shmflg* & *IPC_EXCL*)) is non-zero.

APPLICATION USAGE

The functions *shmctl*, *shmget* and *shmat* and *shmdt* are hardware-dependent and may not be present on all systems. The shared memory routines should not be used by applications except when extreme performance considerations require them.

SEE ALSO

shmctl(2), *shmop*(2), *shm*(5).

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from the entry in Issue 2 of the SVID.

NAME

shmat, shmdt — shared-memory-operations (OPTIONAL)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
char *shmat(shmid, shmaddr, shmflg)
int shmid;
char *shmaddr
int shmflg;
```

```
int shmdt(shmaddr)
char *shmaddr
```

DESCRIPTION

The function *shmat* attaches the shared-memory-segment associated with the shared-memory-identifier specified by *shmid* to the data segment of the calling-process. The segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system.

If *shmaddr* is not equal to zero and (*shmflg* & *SHM_RND*) is non-zero, the segment is attached at the address given by (*shmaddr*-(*shmaddr* % *SHMLBA*)). The character % is the C language modulus operator.

If *shmaddr* is not equal to zero and (*shmflg* & *SHM_RND*) is equal to zero, the segment is attached at the address given by *shmaddr*.

The segment is attached for reading if (*shmflg* & *SHM_RDONLY*) is non-zero and the calling-process has read permission; otherwise, if it is not true and the calling process has read and write permission, the segment is attached for reading and writing.

The function *shmdt* detaches from the calling-process's data segment the shared memory segment located at the address specified by *shmaddr*.

The following symbolic names are defined by the *<sys/shm.h>* header file:

Name	Description
SHMLBA	segment low boundary address multiple
SHM_RDONLY	attach read-only (else read-write)
SHM_RND	round attach address to SHMLBA

RETURN VALUE

If successful, the function *shmat* will return the data segment start address of the attached shared-memory-segment. If successful, the function *shmdt* will return a value of zero. Otherwise, the function *shmat* and the function *shmdt* will return -1 and *errno* will indicate the error.

ERRORS

The function *shmat* will fail and not attach the shared-memory-segment if one or more of the following are true:

[EACCES]

Operation permission is denied to the calling-process, see *ipc(2)*.

[ENOMEM]

The available data space is not large enough to accommodate the shared memory segment.

[EINVAL]

The value of *shmid* is not a valid shared-memory-identifier; or the value of *shmaddr* is not equal to zero and the value of $(shmaddr - (shmaddr \% SHMLBA))$ is an illegal address; or the value of *shmaddr* is not equal to zero, $(shmflg \& SHM_RND)$ is equal to zero and the value of *shmaddr* is an illegal address.

[EMFILE]

The number of shared-memory-segments attached to the calling-process would exceed the system-imposed limit.

The function *shmdt* will fail and not detach the shared-memory-segment if the following is true:

[EINVAL]

Shmaddr is not the data segment start address of a shared-memory-segment.

APPLICATION USAGE

The functions *shmctl*, *shmget*, *shmat*, and *shmdt* are hardware dependent and may not be present on all systems. The shared memory routines should not be used by applications except when extreme performance considerations require them.

SEE ALSO

exec(2), *exit(2)*, *fork(2)*, *ipc(2)*, *shmctl(2)*, *shmget(2)*.

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from the entry in Issue 2 of the SVID.

NAME

ipc — inter-process communication access structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
```

DESCRIPTION

There are three mechanisms for inter-process communication (IPC): messages, semaphores and shared memory. All use a common structure type, *ipc_perm* to pass information used in determining permission to perform an IPC operation.

The structure *ipc_perm* contains the following members:

```
    ushort    uid;        /* owner's user ID */
    ushort    gid;        /* owner's group ID */
    ushort    cuid;       /* creator's user ID */
    ushort    cgid;       /* creator's group ID */
    ushort    mode;       /* read/write permission */
```

Definitions are given for the following constants:

Mode bits:

```
    IPC_CREAT  /* create entry if key doesn't exist */
    IPC_EXCL   /* fail if key exists */
    IPC_NOWAIT /* error if request must wait */
```

Keys:

```
    IPC_PRIVATE /* private key */
```

Control Commands:

```
    IPC_RMID /* remove identifier */
    IPC_SET  /* set options */
    IPC_STAT /* get options */
```

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from UNIX System V, Release 2.

NAME

msg — message queue structures

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

DESCRIPTION

The following constants and structure members are defined:

Message operation flags:

MSG_NOERROR /* no error if big message */

The structure *msqid_ds* contains the following members:

struct ipc_perm	msg_perm;	/* operation permission structure */
ushort	msg_qnum;	/* number of messages currently on queue */
ushort	msg_qbytes;	/* max number of bytes allowed on queue */
ushort	msg_lspid;	/* pid of last msgsnd */
ushort	msg_lrpid;	/* pid of last msgrcv */
time_t	msg_stime;	/* time of last msgsnd */
time_t	msg_rtime;	/* time of last msgrcv */
time_t	msg_ctime;	/* time of last change */

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from UNIX System V, Release 2.

NAME

sem — semaphore facility

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

DESCRIPTION

The following constants and structures are defined:

Semaphore operation flags:

```
SEM_UNDO    /* set up adjust on exit entry */
```

Semctl command definitions:

```
GETNCNT    /* get semncnt */
GETPID      /* get sempid */
GETVAL      /* get semval */
GETALL      /* get all semvals */
GETZCNT     /* get semzcnt */
SETVAL      /* set semval */
SETALL      /* set all semvals */
```

The structure *semid_ds* contains the following members:

```
struct      ipc_perm  sem_perm;    /* operation permission struct */
ushort      sem_nsems; /* number of semaphores in set */
time_t      sem_otime; /* last semop time */
time_t      sem_ctime; /* last time changed by semctl */
```

The number of semaphores in a set is *sem_nsems*; within the set semaphores number from 0 to *sem_nsems*-1. The number of a semaphore is known as a *sem_num*.

A semaphore is represented by an anonymous structure containing the following members:

```
ushort      semval;    /* semaphore value */
ushort      sempid;    /* pid of last operation */
ushort      semncnt;   /* no of processes waiting for semval to become
                        greater than current value */
ushort      semzcnt;   /* number of processes waiting for semval to become
                        zero */
```

The structure *sembuf* contains the following members:

```
ushort      sem_num;   /* semaphore number */
short       sem_op;    /* semaphore operation */
short       sem_flg;   /* operation flags */
```

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from UNIX System V, Release 2.

NAME

shm — shared memory facility

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

DESCRIPTION

The following constants and structures are defined:

Message operation flags:

```
SHM_RDONLY    /* attach read-only (else read-write) */
SHMLBA        /* segment low boundary address multiple */
SHM_RND        /* round attach address to SHMLBA */
```

The structure *shmid_ds* contains the following members:

```
struct ipc_perm shm_perm;    /* operation permission structure */
int shm_segsz;    /* size of segment in bytes */
ushort shm_lpid;    /* pid of last shmop */
ushort shm_cpid;    /* pid of creator */
ushort shm_nattch;    /* number of current attaches */
time_t shm_atime;    /* time of last shmat */
time_t shm_dtime;    /* time of last shmdt */
time_t shm_ctime;    /* time of last change by shmctl */
```

CHANGE HISTORY

First released in Issue 2 of the Guide.

Issue 2

Derived from UNIX System V, Release 2.

X/O/P/E/N/

PORTABILITY GUIDE

THE X/OPEN SYSTEM V SPECIFICATION
SOURCE CODE TRANSFER



Contents

Chapter	1	SOURCE CODE TRANSFER
Chapter	2	FLOPPY DISC
	2.1	INTRODUCTION
	2.2	PHYSICAL RECORDING
	2.3	SPECIAL FILE NAMES
	2.4	DEVICE NUMBERS
	2.5	MEDIA
Chapter	3	MAGNETIC TAPE
	3.1	INTRODUCTION
	3.2	PHYSICAL RECORDING
	3.3	SPECIAL FILE NAMES AND BLOCKING
	3.4	DEVICE NUMBERS
Chapter	4	UTILITIES
	4.1	INTRODUCTION
	4.2	BYTE ORDERING
	4.3	CPIO
	4.4	TAR
Chapter	5	OTHER TECHNIQUES
	5.1	UUCP

Source Code Transfer

One of the major problems inhibiting the porting of applications between UNIX system derivatives is that of incompatible media standards and the physical problems of transferring source code in machine readable form.

This Part of the Guide includes X/OPEN definitions for the transfer of source code, including portable media formats and guidelines in the use of transfer utilities. The physical characteristics and necessary formatting information are described, as are the special file names by which these devices are known in X/OPEN systems.

Standards are defined for transfer of 5 ¼ inch floppy discs and ½ inch magnetic tapes between machines. Because of the different nature of X/OPEN systems, ranging from single-user work stations to large mainframes, it is not possible to define a single portable medium which is supported across the whole range. Defining standards for both floppy discs and ½ inch magnetic tape gives the highest practical coverage of systems.

Current differences in the physical recording formats between cartridge tape devices prevents the definition of a standard for this popular medium.

Floppy Disc

2.1 INTRODUCTION

As exchange media, the X/OPEN Group defines formats for 40 and 80 track floppy discs.

The prime format is 80 track, conforming with the standard ECMA-78, track format 2. The 40 track format, conforming with the standard ECMA-70, track format 2, is defined for compatibility with personal computers.

X/OPEN systems equipped only with 80 track disc drives may offer the facility to read 40 track discs.

2.2 PHYSICAL RECORDING

The standard floppy disc recording formats are shown below:

Floppy Disk Recording Formats
80 tracks (96 tracks per inch) on each side 2 tracks per cylinder 9 sectors per track 512 bytes per sector Modified Frequency Modulation (MFM) recording on all the tracks Double Sided - Double Density Conformity with standard ECMA-78, track format 2
40 tracks (48 tracks per inch) on each side 2 tracks per cylinder 9 sectors per track 512 bytes per sector Modified Frequency Modulation (MFM) recording on all the tracks Double Sided - Double Density Conformity with standard ECMA-70, track format 2

Note that the 80 track format is preferred; systems equipped only with 80 track disc drives may be able to read but not write 40 track discs.

2.3 SPECIAL FILE NAMES

The special file names associated with these formats are:

Number of Tracks	Name
40	/dev/sctfdl<number>
80	/dev/sctfdm<number>

The device names are usually links to system-specific device names.

2.4 DEVICE NUMBERS

The part of the special file name described in the table above as <number> is constructed from a system specific unit number. To this is added 0 or 128 (decimal) to define whether cylinder 0 is accessible. 0 means that cylinder 0 is accessible, so the first sector accessed is sector 1 track 0 cylinder 0. 128 means that cylinder 0 is not accessible, so the first sector accessed is sector 1 track 0 cylinder 1.

To conform with ECMA 107 standards, cylinder 0 should only be used for administrative information and the source code files should start in sector 1, track 0, cylinder1.

2.5 MEDIA

There is no standard method of handling media flaws (for example, alternative track recording). The X/OPEN definition therefore requires that the transfer medium must be error free.

Magnetic Tape

3.1 INTRODUCTION

The X/OPEN standard for magnetic tape covers ½ inch magnetic tape with a number of different recording formats and densities. The "preferred" format is 9 track phase encoded (PE) at 1600 bits per inch.

3.2 PHYSICAL RECORDING

The preferred physical tape recording format is:

- 9 track Phase Encoded (PE), 1600 bits per inch (bpi).

Optional formats that may also be supported by particular systems in addition to this are

- 9 track Group Code Recording (GCR), 6250 bpi.
- 9 track Non Return to Zero Inverted (NRZI), 800 bpi.

3.3 SPECIAL FILE NAMES AND BLOCKING

The device names associated with these formats are:

name	format	blocksize(bytes)
/dev/sctmtl<number>	NRZI	512
/dev/sctmtm<number>	PE	512
/dev/sctmth<number>	GCR	512
/dev/rsctmtl<number>	NRZI	see below
/dev/rsctmtm<number>	PE	see below
/dev/rsctmth<number>	GCR	see below

On the "raw" devices (rsct...), data is both read and written in blocks corresponding to the length requested in the read or write system call; see *read(2)* and *write(2)* in "SYSTEM CALLS AND LIBRARIES".

The device names are usually links to system-specific device names.

3.4 DEVICE NUMBERS

The part of the special file name described in the table above as <number> is constructed from a system-specific unit number with the addition of 0 or 128 (decimal) to indicate whether the tape is to be rewound on closure. Any tape that is opened for writing has a tape mark written on closure. Addition of 0 to the unit number causes the tape to be rewound to the beginning of tape mark (BOT); addition of 128 inhibits this.

Hosted implementations may need extra information to be specified in the device names, for example volume names.

4.1 INTRODUCTION

X/OPEN systems support two standard utilities for use with any of the devices mentioned in this part of the Guide. *Cpio* is the preferred utility; *tar* is included primarily because certain versions of the UNIX operating system only support that form.

If an exchange medium is to be read on a target machine that is architecturally different from the source machine, problems may arise concerning the ordering of bytes within a word and words within a long word (see the portability guidelines in "C LANGUAGE"). These can easily be handled when using *cpio* as the exchange utility, while with *tar* it may be a little more difficult.

4.2 BYTE ORDERING

An array of bytes should be written to the physical medium as if consecutive single bytes were written.

4.3 CPIO

This is the preferred utility for source code transfer.

The `—c` option is mandatory to prevent the occurrence of byte ordering problems.

The `—B` option, which enables 5120 byte blocking, is only effective if it is used on a "raw" tape device (`/dev/rsctmt...`), where it helps to reduce the amount of tape wasted on inter-record gaps. It has no effect on other devices.

The following conventions must be used:

- All the pathnames and the name "TRAILER!!!" are represented in US ASCII (ISO code 646), using 8-bit characters without parity.
- The maximum value for a user or a group ID {UID-MAX} is 32000 (see *limits(5)*).
- All the *cpio* archive is stored on a single magnetic storage medium.
- There is only one *cpio* archive stored per magnetic storage medium.
- Relative pathnames are used (i.e., no leading `/`).

On a magnetic tape, the following conventions must be used:

- The magnetic tape must be written in "raw" mode.
- The record length must be 5 120 bytes (see option `—B` in `cpio(1)`).

4.4 TAR

This utility may be used, but it is not recommended unless it is necessary to read, for example, media written elsewhere using `tar`. It should only be used to write 512 byte records or 10 240 byte records (with the `—b 20` option). Other record sizes may cause problems when other systems try to read the archive produced.

Other Techniques

5.1 UUCP

Among other methods of moving data between X/OPEN systems, direct machine to machine connection using *uucp* and its utilities is the most relevant, since it is supplied with most X/OPEN systems. However, *uucp* is far from easy to use and the following points should be noted:

- The systems to be connected must have a compatible serial interface, preferably V.24/RS232C. Not all suppliers support a reasonable subset of the V.24 specification and in particular, many of its signalling lines are not supported. However, these lines are not essential as long as the configuration file, *L.sys*, is set up appropriately. Baud rate, character size and parity have to be configurable.
- The asynchronous line drivers should support XON/XOFF (ASCII DC1/DC3) handshaking and an 8-bit transparent mode.
- The interface should provide enough internal buffering to avoid over-runs at higher baud rates. Otherwise, retransmissions will make data transfer time excessive.
- The versions of *uucp* must be compatible.

Besides this most common form of networking between X/OPEN systems, other networks may be available. Care must be taken in their use, since the higher level protocols often differ from system to system.

See *uucp*(1) in "XVS COMMANDS AND UTILITIES".

ISBN: 0 444 70176 1